МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВЕТЕРИНАРНОЇ МЕДИЦИНИ ТА БІОТЕХНОЛОГІЙ ІМЕНІ С.З. ҐЖИЦЬКОГО
ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

другого (магістерського) рівня вищої освіти

на тему:

ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ ФРЕЙМВОРКІВ ПРИ СТВОРЕННІ КЛІЄНТСЬКОЇ ЧАСТИНИ САЙТУ

Виконав: студент групи ІТ-61
спеціальності 126 «Інформаційні системи та технологій»
Бойко І. М.
(прізвище та ініціали)

Керівник: Желєзняк А.М.
(прізвище та ініціали)

ДУБЛЯНИ 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВЕТЕРИНАРНОЇ МЕДИЦИНИ ТА БІОТЕХНОЛОГІЙ ІМЕНІ С.З. ҐЖИЦЬКОГО
ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Рівень вищої освіти другий (магістерський)
Спеціальність 126 «Інформаційні системи та технології»

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____
(підпис)
д.т.н., професор, Тригуба А. М.
(вч. звання, прізвище, ініціали)
“____” _____202 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Бойко Іллі Миколайовича
(прізвище, ім’я, по батькові)

1. Тема роботи *«Дослідження ефективності використання фреймворків при створенні клієнтської частини сайту»*
керівник роботи *к. е н., доцент., Желєзняк А.М.*
(наук.ступінь, вч. звання, прізвище, ініціали)

затверджені наказом ЛНУВМБТ ім.С.З.Гжицького №140/к-с від 28.02.2025

2. Строк подання студентом роботи 05.12.2025 р.
3. Вихідні дані: *аналітичні дані роботи та характеристика об’єкту дослідження, опис бібліотек мов програмування, науково-технічна і довідкова література.*
4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)
Вступ
1. Аналіз стану питання в теорії та практиці
2. Обґрунтування, вибір та реалізація інструментарію вирішення задачі
3. Результати вирішення задачі
4. Охорона праці та безпека в надзвичайних ситуаціях
5. Визначення ефективності
Висновки
Бібліографічний список
5. Перелік графічного матеріалу
Графічний матеріал подається у вигляді презентації

6. Консультанти розділів

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата		Відмітка про виконання
		завдання видав	завдання прийняв	
1, 2, 3, 5	*Желєзняк А.М., к.е.н., доцент кафедри інформаційних технологій*			
4	*Городецький І.М., к.т.н., доцент кафедри інженерної механіки*			

7. Дата видачі завдання 01.03.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Відмітка про виконання
1	*Отримання завдання. Вивчення літератури по темі роботи. Написання першого розділу*	*01.03.2025 – 31.05.2025*	
2	*Проектування та опис технічного завдання, обґрунтування та вибір інструментарію реалізації проекту (написання другого розділу).*	*01.06.2025 – 31.08.2025*	
3	*Програмна реалізація поставленого завдання (написання третього розділу)*	*01.09.2025 – 17.10.2025*	
4	*Написання розділу з охорони праці*	*18.10.2025 – 04.11.2025*	
5	*Оцінка ефективності поставленого завдання (виконання п'ятого розділу)*	*05.11.2025 – 18.11.2025*	
6	*Завершення оформлення основної частини, написання висновків та підготовка презентаційного матеріалу*	*19.11.2025 – 01.12.2025*	
7	*Завершення роботи в цілому. Підготовка до захисту кваліфікаційної роботи*	*02.12.2025 – 08.12.2025*	

Студент _____ *Бойко І.М.*
(підпис) (прізвище та ініціали)

Керівник роботи _____ *Желєзняк А.М.*
(підпис) (прізвище та ініціали)

УДК 004.451.3:004.43

Дослідження ефективності використання фреймворків при створенні клієнтської частини сайту.

Бойко Ілля Миколайович. Кваліфікаційна робота. Кафедра інформаційних технологій. Дубляни, Львівський національний університет ветеринарної медицини та біотехнологій ім.С.З.Гжицького, 2025 р.

Кваліфікаційна робота: 63 сторінки текстової частини, 8 таблиць, 18 рисунків, 24 джерела літератури, 7 додатків.

Подано теоретичні основи розвитку фреймворків, які використовуються в веб-розробці. Проаналізовано предметну область, існуючі аналоги та визначено методичні підходи до їх вибору. Обґрунтовано вибір технологічного стеку для розробки клієнтської частини сайту проекту.

Запропоновано поєднання використання фреймворків Vue 3, Nuxt 3 та CSM Strapi для реалізації веб-застосунку. Здійснено реалізацію проекту та визначені перспективи його розвитку на основі запропонованих метрик.

Здійснено аналіз травматичних ситуацій при виконанні різних робіт у сфері використанням комп’ютерної техніки, викладено питання охорони праці.

Ключові слова: сайт, веб-розробка, фреймворки, клієнтський інтерфейс

ЗМІСТ

ВСТУП

Сучасна веб-розробка переживає період інтенсивної еволюції, що зумовлено зростанням вимог до продуктивності, масштабованості та інтерактивності веб-застосунків. Традиційні підходи до створення клієнтських інтерфейсів уже не забезпечують необхідного рівня гнучкості та швидкості розробки, тому провідні технологічні компанії та інженери переходять до використання високорівневих фреймворків. Такі інструменти як Vue.js, React, Angular чи Svelte, а також метарамки (meta-frameworks) на зразок Nuxt чи Next, пропонують структурований спосіб організації коду, ефективну роботу з даними та значно спрощене управління життєвим циклом застосунку.

Актуальність теми полягає у необхідності порівняння класичних підходів до веб-розробки з сучасними фреймворками та обґрунтуванням того, як саме їх використання впливає на продуктивність, якість коду, швидкість розробки та кінцевий досвід користувача.

Кваліфікаційна робота присвячена аналізу застосування веб-фреймворків у реальному проєкті, зосередженому на екосистемі Vue.js та Nuxt3. В роботі реалізовано повноцінну веб-систему з компонентною архітектурою, серверним рендерингом, оптимізованими механізмами кешування та налаштовано маршрутизацію. Такий практичний контекст дозволяє не лише проаналізувати теоретичні особливості фреймворків, але й довести їх ефективність через результати реальної імплементації.

Метою кваліфікаційної роботи є обґрунтування ефективності застосування веб-фреймворків у сучасній веб-розробці та демонстрація цієї ефективності на прикладі розробки реального застосунку за допомогою Vue.js та Nuxt 3.

Для досягнення цієї мети передбачається виконання таких завдань:

- здійснити огляд теоретичних основ, наукових досліджень та історії розвитку веб-фреймворків, особливостей їх застосування у веб-розробці;

- здійснити порівняльний аналіз найбільш популярних фреймворків, таких як Vue, React та Angular;
- обґрунтувати вибір технологічного стеку Vue.js + Nuxt 3 для практичної реалізації проекту;
- дослідити результати реалізації проєкту та проаналізувати ефективність застосування фреймворків на основі запропонованих метрик та підходів;
- визначити перспективи подальшого розвитку архітектури проекту та можливостей її оптимізації.

Кваліфікаційна робота складається з п’яти розділів. В першому з них наведено теоретичні основи веб-розробки та проаналізовано еволюцію й особливості сучасних JavaScript-фреймворків. Другий розділ містить обґрунтування вибору архітектури та технологічного стеку, зокрема Vue.js, Nuxt 3 та Strapi для формування контенту клієнтської частини сайту. У третьому розділі представлено результати практичної реалізації проєкту, включно з побудовою компонентної системи, API-взаємодії та ключових елементів архітектури. Четвертий розділ присвячено питанням охорони праці та безпеки в надзвичайних ситуаціях. П’ятий розділ містить оцінку ефективності використання фреймворків у контексті виконаної роботи.

Наукова новизна полягає у комплексному аналізі ефективності застосування сучасних веб-фреймворків на прикладі реального проєкту, що поєднує компонентний підхід, серверний рендеринг та інтеграцію з headless-CMS. Робота демонструє практичну цінність використання Vue.js та Nuxt 3 для побудови масштабованих і продуктивних веб-систем та формує підхід, який може бути адаптований у подальших розробках. Результати дослідження апробовані на VII Всеукраїнської науково-практичної конференції “Інформаційна безпека та інформаційні технології” (ІБІТ – 2025), 27 листопада 2025 р., м.Львів із публікацією тез.

РОЗДІЛ 1.
АНАЛІЗ СТАНУ ПИТАННЯ В ТЕОРІЇ ТА ПРАКТИЦІ

1.1 Теоретичні основи розробки клієнтської частини у веб-розробці

Клієнтська частина веб-застосунків відіграє ключову роль у забезпеченні взаємодії користувача з програмною системою. Основне завдання frontend-рівня полягає у формуванні інтерфейсу, виконанні логіки відображення та забезпечення коректної роботи застосунку в браузері незалежно від пристрою, швидкості мережі чи особливостей операційної системи. Теоретичні основи побудови клієнтської частини спираються на низку концепцій, що визначають архітектуру та поведінку сучасних веб-інтерфейсів.

Провівши детальний аналіз наукових досліджень та публікацій за темою кваліфікаційної роботи [1-3], було зроблено узагальнення, що під клієнтською частиною веб-застосунку (front-end) можна розуміти програмну складову веб-застосунку, яка виконується на стороні користувача у веб-браузері та забезпечує візуальне відображення інформації, обробку дій користувача, керування станом інтерфейсу і взаємодію із серверною частиною через мережеві протоколи з метою надання інтерактивного та зручного користувацького інтерфейсу.

Одним із базових принципів є розподіл відповідальності між рівнями системи. У клієнтській частині зосереджені функції відображення даних, обробки подій, управління локальним станом та часткової логіки взаємодії. Це дозволяє зменшити навантаження на сервер і забезпечити високу інтерактивність інтерфейсу, що є критично важливим для сучасних веб-застосунків.

Аналіз існуючих практик та наукових досліджень [4,5] показав, що існують різні підходи у розробці клієнтської частини сайту. Одним із них є рендеринг на стороні клієнта.

Загалом рендеринг на стороні клієнта (Client-side rendering (CSR)) - це підхід у веб-розробці, на основі якого браузер користувача (клієнт) використовує JavaScript для генерації та відображення основного контенту веб-сторінки (рис.1.1.).

Сервер передає клієнту базову HTML-структуру сторінки разом із набором JavaScript-файлів, після чого веб-браузер виконує отриманий код, асинхронно завантажує необхідні дані та динамічно формує вміст сторінки [6].

Такий підхід забезпечує високий рівень інтерактивності, швидку реакцію інтерфейсу на дії користувача та зменшення кількості повних перезавантажень сторінок, однак водночас може ускладнювати початкову індексацію веб-ресурсу пошуковими системами та вимагати застосування додаткових механізмів оптимізації для SEO.

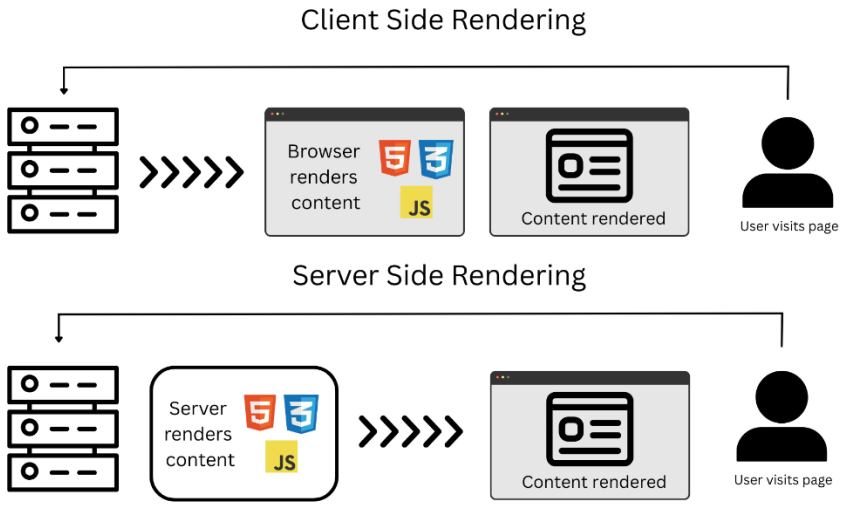

Рисунок 1.1. – Рендеринг на стороні сервера та клієнта [6]

Застосування підходу у веб-розробці, що передбачає рендеринг на стороні сервера базується на тому, що основна робота виконується на стороні сервера, який генерує повний HTML-код сторінки, перш ніж надсилати її до браузера.

Важливим аспектом теорії розробки клієнтської частини веб-застосунків є компонентний підхід, який передбачає поділ інтерфейсу на незалежні блоки та модулі. Кожен компонент містить шаблон, стилі та логіку, що підвищує

зручність повторного використання коду та спрощує супровід системи. Цей підхід став фундаментом для сучасних фреймворків, включно з Vue, який використовується в практичній частині даної кваліфікаційної роботи.

Таблиця 1.1. – Приклади компонентного підходу у веб-розробці

Назва бібліотеки/ фреймворку	Опис реалізації компонентного підходу	Приклади компонент
React	Користувацький інтерфейс будується з функціональних або класових компонентів. Кожен компонент повертає JSX-розмітку та може отримувати дані через властивості і зберігати локальний стан.	Компоненти Header, Footer, ProductCard, які об’єднуються в компонент Page.
Angular	У фреймворку Angular компонент є основною будівельною одиницею застосунку та складається з шаблону, класу та метаданих. Компоненти організовуються в модулі, що спрощує масштабування великих проєктів.	Компонент списку курсів, компонент детального перегляду курсу, компонент форми реєстрації
Vue.js	У Vue.js використовується підхід однофайлових компонентів (Single File Components), у яких HTML, JavaScript і CSS об’єднані в одному файлі. Це підвищує читабельність і зручність підтримки коду.	Компонент навігаційної панелі, компонент модального вікна, компонент таблиці даних

Компонентний підхід є фундаментальною концепцією сучасної веб-розробки, яка сприяє створенню масштабованих, гнучких і зручних у супроводі клієнтських застосунків. Саме завдяки цьому підходу більшість сучасних клієнтських фреймворків забезпечують ефективну організацію коду та підвищують продуктивність роботи розробників.

В наукових дослідженнях на основі структури компоненти поділяють на два класи [8]: повторно використовувані (динамічні) та статичні компоненти.

До повторно використовуваних компонентів відносяться компоненти, що формують зовнішню структуру веб-застосунку, зокрема заголовки (header), бічні панелі навігації (sidebar) та нижні колонтитули (footer). Зазвичай вони зберігають сталу структуру та вигляд на більшості сторінок застосунку. Такі компоненти забезпечують єдність інтерфейсу та спрощують підтримку візуального стилю.

Динамічні компоненти використовуються у випадках, коли основний вміст сторінки може змінюватися залежно від обраної послуги, пункту меню або сценарію використання. Ці компоненти відповідають за відображення змінних даних і логіки та підвантажуються відповідно до контексту або маршруту користувача на сайті.

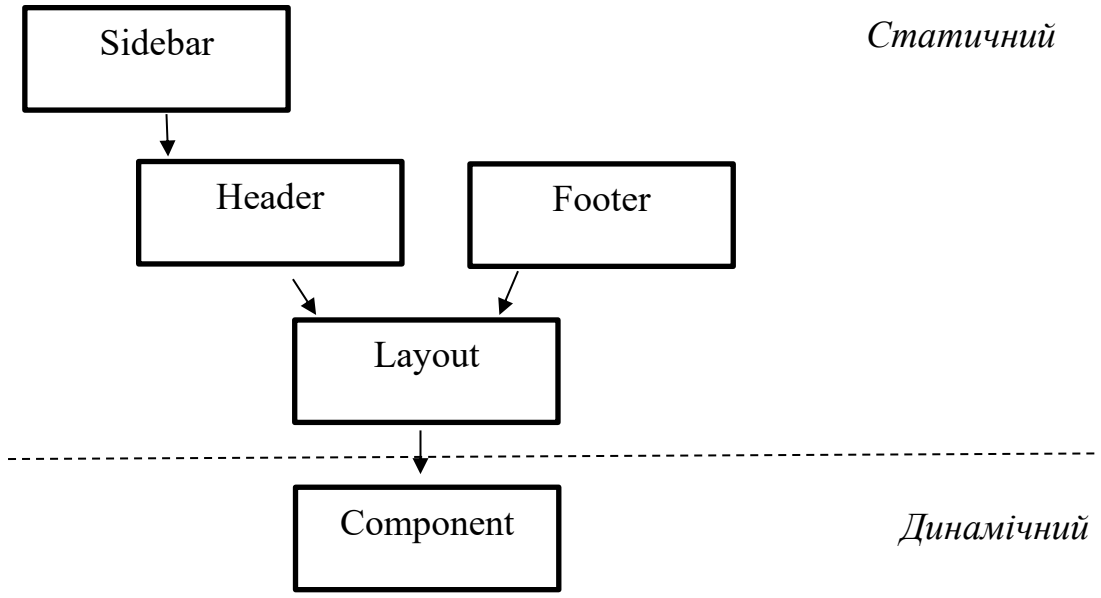

Рисунок 1.2. – Структура клієнтського інтерфейсу на основі компонентного підходу [8]

Концепція компонентів макета (layout components) передбачає поєднання статичних елементів інтерфейсу та змінного вмісту. Компонент макету сторінки заздалегідь містить структуру заголовка, бічної панелі та

нижнього колонтитулу і фактично є шаблоном, у межах якого відображається динамічний вміст клієнтської . Такий підхід дозволяє чітко розмежувати відповідальність між компонентами, підвищити повторне використання коду, а також спростити масштабування та модифікацію клієнтської частини веб-застосунку.

Не менш важливою концепцією у веб-розробці є реактивність, що дозволяє автоматично синхронізувати стан даних із відображенням клієнтського інтерфейсу. У традиційній моделі DOM-маніпуляцій оновлення відбувається вручну, що ускладнює створення складних інтерфейсів. Сучасні фреймворки використовують реактивні системи, що оптимізують оновлення UI та підвищують стабільність роботи застосунку. В таблиці 1.2 наведено основні принципи реактивного програмування.

Таблиця 1.2 – Принципи реактивного програмування та їх опис.

Назва принципу	Короткий опис
Потокова природа даних	Будь-яка подія (натискання кнопки, введення тексту, відповідь від сервера тощо) розглядається як елемент потоку. Потоки можуть бути нескінченними, комбінуватися між собою, фільтруватися або трансформуватися.
Спостереження та підписка	Компоненти або функції не запитують дані безпосередньо, а підписуються на потоки подій. При появі нових значень система автоматично повідомляє всіх підписників, що забезпечує декларативний стиль програмування.
Асинхронність і неблокуюча обробка	Потоки подій обробляються асинхронно, що дозволяє застосунку ефективно працювати з мережевими запитами, таймерами та користувацькими подіями без блокування основного потоку виконання.
Композиція потоків	Реактивне програмування надає механізми для об’єднання, трансформації та синхронізації кількох потоків подій. Це дає змогу будувати складну логіку обробки подій із простих елементів.

Під реактивним програмуванням можна в загальному розуміти парадигму розробки програмного забезпечення, яка орієнтована на роботу з асинхронними потоками даних і подій, забезпечуючи автоматичне реагування системи на їх зміни. Реактивний підхід дозволяє розробнику ефективно обробляти численні події, що виникають у клієнтській частині веб-застосунку, такі як дії користувача, мережеві запити, таймери, зміни стану інтерфейсу або даних. Одним із ключових понять в реактивному програмуванні є потік подій (рис.1.3).

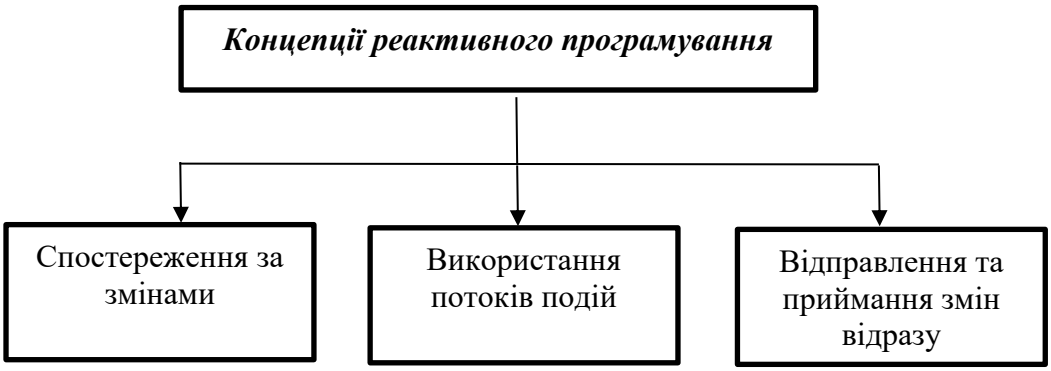

Рисунок 1.3 – Концепції підходу реактивного програмування у веб-розробці

В цілому під потоком подій (event stream) можна розуміти послідовність подій або значень, що виникають у часі та можуть бути асинхронно оброблені, а бізнес-логіка застосунку будується навколо опису того, як система має реагувати на появу нових подій або зміну даних. Потоки подій під час розробки клієнтської частини сайту можуть формуватися з різних джерел, зокрема на основі події DOM (кліки, скролінг, введення тексту); реалізації HTTP-запитів та відповідей серверів; використання таймерів та інтервалів; зміни стану застосунку тощо.

Frontend також ґрунтується на принципах маршрутизації та управління переходами між сторінками без повного перезавантаження документа. Це забезпечує плавну взаємодію та дає змогу будувати односторінкові застосунки (SPA), які є стандартом у сучасній веб-розробці. Також у контексті Nuxt 3

маршрутизація поєднується з можливістю серверного рендерингу, що розширює функціональні можливості системи.

Окремого значення набувають серверний рендеринг (SSR) та гідрація, які формують підхід до створення універсальних веб застосунків. SSR дозволяє зменшити час першого відображення сторінки, покращує SEO та робить застосунок доступнішим для пошукових систем. Гідрація забезпечує коректне підключення клієнтської логіки після початкового рендерингу, поєднуючи продуктивність серверного рендерингу з гнучкістю SPA.

У сукупності ці теоретичні аспекти – компонентність, реактивність, маршрутизація, управління станом та застосування SSR – формують основу клієнтської частини сучасних веб-застосунків та визначають ефективність роботи фреймворків, що використані у практичній частині дослідження за темою кваліфікаційної роботи.

1.2 Розвиток веб-фреймворків для створення клієнтської частини сайту

Історія розвитку веб-фреймворків безпосередньо пов’язана з еволюцією веб-технологій та зростанням вимог до динамічності інтерфейсів. Перші веб-сторінки були статичними, а їхній функціонал обмежувався HTML і мінімальними стилями. З появою JavaScript у 1995 році стало можливим виконувати код на стороні клієнта, однак ранні підходи ґрунтувалися на прямому маніпулюванні DOM, що робило процес розробки складним і маломасштабованим.

Починаючи з 2000-х років з’явилися перші інструменти, які спрощували роботу з DOM і забезпечували базову структуру для створення інтерфейсів. Одним із найвідоміших став jQuery, що пропонував полегшену модель роботи з елементами сторінки, подіями та AJAX-запитами. Проте він не вирішував головної проблеми – відсутності чіткої архітектури застосунків.

З часом зросла потреба у структурованому підході, що спричинило появу фреймворків із чіткою моделлю розробки. У 2010 році з’явився

Knockout.js, який уперше запровадив підхід MVVM та реактивні прив'язки даних між моделями й інтерфейсом. У тому ж році компанія Google випускає AngularJS, який запровадив декларативні шаблони з двосторонньою прив'язкою даних. На рис.1.4. представлено схему архітектури MVVM, яка демонструє розділення даних, логіки представлення та користувацького інтерфейсу в клієнтській частині застосунку.

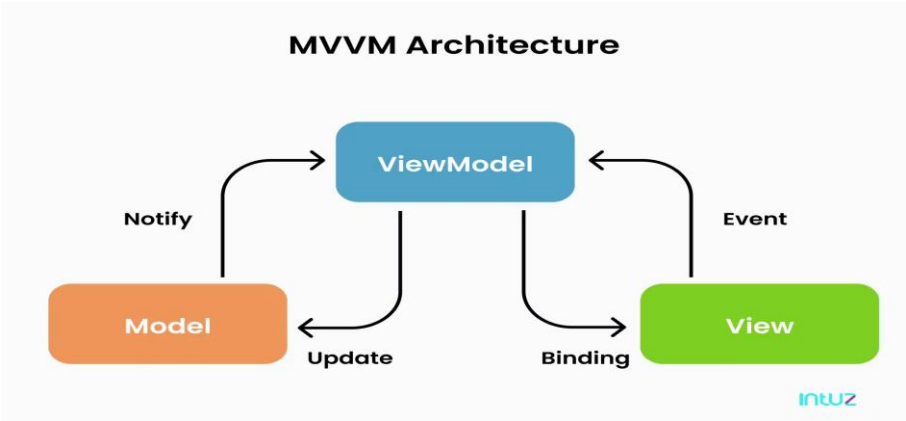

Рисунок 1.4 - Схема архітектури MVVM (Model-View-ViewModel) [13]

Подальший розвиток Angular привів до появи Angular2+, який отримав повністю нову компонентну архітектуру та перехід на TypeScript. Компанія Facebook випустила React у 2013 році, який в цей час представив компонентну модель та віртуальний DOM, що стали стандартом індустрії.

На рік пізніше світ побачив Vue.js, який об'єднав ключові ідеї Angular і React, додав простий синтаксис та гнучку архітектуру. Після виходу початкової версії Vue з'явились Vue 2, а згодом і Vue 3, запровадивши Composition API та ще більш сучасну реактивну систему. У цей же період з'явився й Svelte, який запровадив нетиповий підхід – компіляцію компонентів у високопродуктивний JavaScript без використання віртуального DOM.

Ці фреймворки вирішили проблеми організації коду, реактивності та масштабованості, проте залишили розробникам складні питання маршрутизації, SSR, оптимізації бандлів і структури проєктів.

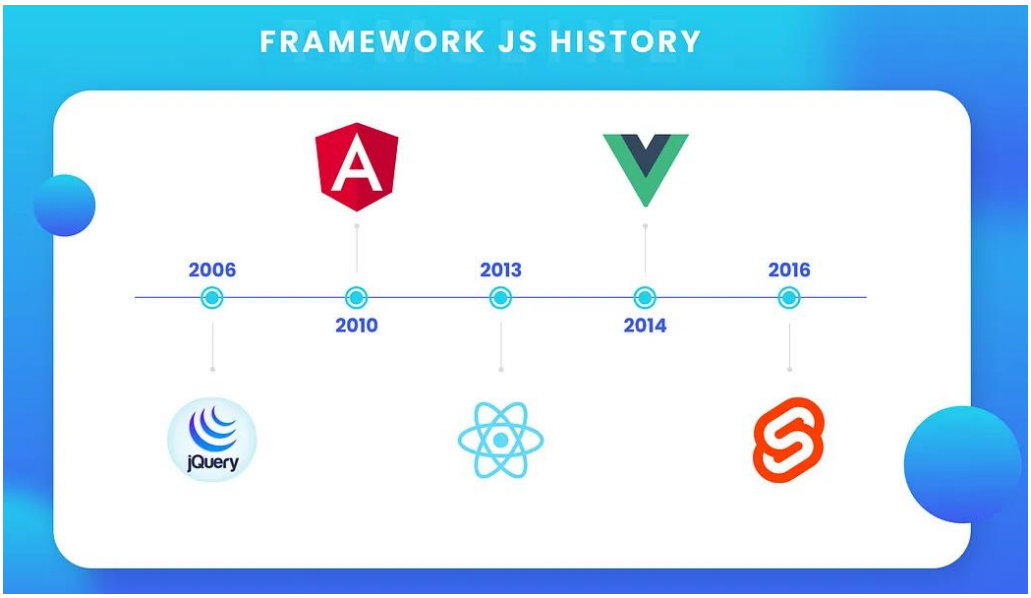

Рисунок 1.5 - Хронологія появи перших версій основних JavaScript-фреймворків [14]

Починаючи з 2016 року у сфері веб-розробки почали з’являтись метафреймворки. Вони створили рішеннями які не лише допомагали будувати інтерфейс, а й надавали можливість реалізовувати повноцінну архітектуру в застосунках. Такі інструменти отримали назву метафреймворків, оскільки вони стоять над базовими фреймворками і доповнюють його серверними можливостями. Аналіз літературних джерел та інтернет-ресурсів [10-12] дозволяє виділити наступні характеристики метафреймворків:

1) використання існуючого фреймворку як бази (наприклад, React або Vue);

2) наявність вбудованої файлової маршрутизації;

3) підтримка різних стратегій рендерингу (CSR, SSR, SSG, ISR);

4) автоматична оптимізація ресурсів (код-сплітинг, кешування, мінімізація);

5) чітко визначена структура проєкту та правила організації компонентів.

Загалом метафреймворки - це надбудови над базовими клієнтськими або серверними фреймворками, які розширили їхню функціональність і надали

структуровані архітектурні рішення, інструменти та домовленості (conventions) для створення повноцінних веб-застосунків. Вони не замінюють базовий фреймворк, а використовують його як ядро, стандартизуючи підхід до організації коду, маршрутизації, рендерингу сторінок і оптимізації продуктивності.

Метафреймворки вирішили ключові проблеми класичних фреймворків:

- складність налаштування SSR;
- відсутність універсальної архітектури;
- ручне керування оптимізацією коду;
- неоднорідність структури проєктів у різних командах.

У контексті клієнтської веб-розробки метафреймворки орієнтовані на вирішення типових проблем сучасних веб-застосунків, зокрема пошукову оптимізацію, швидкість першого завантаження сторінки, керування станом і взаємодію з сервером. Вони поєднують можливості клієнтського рендерингу з серверними підходами, такими як серверний рендеринг (SSR) або генерація статичних сторінок (SSG).

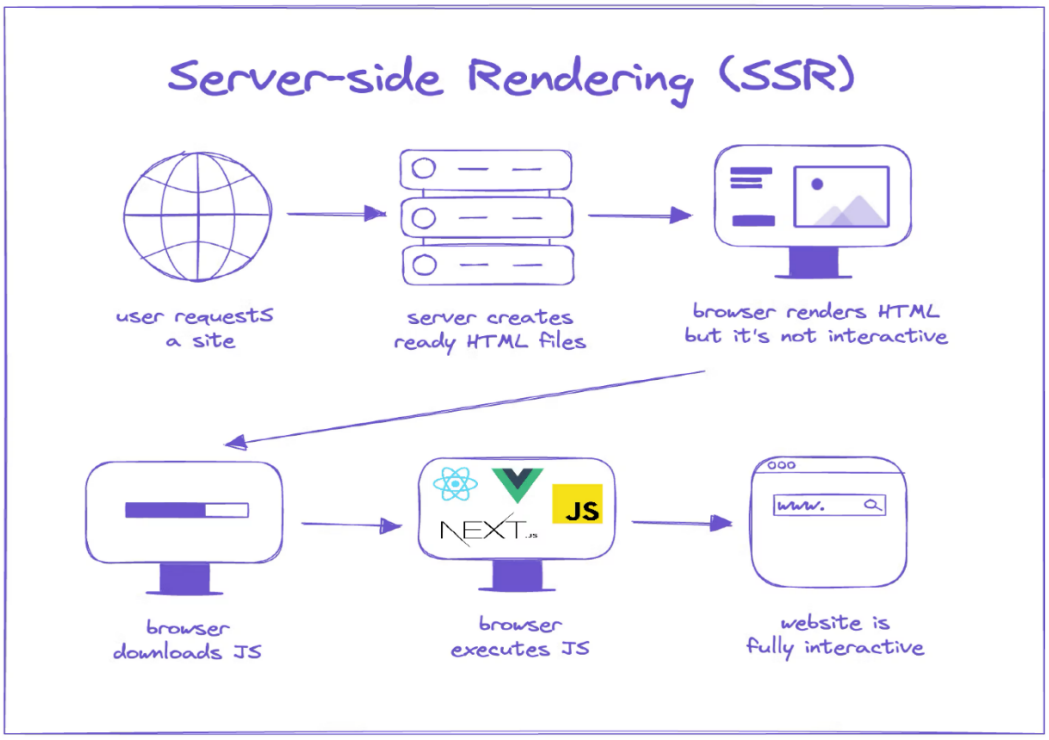

Рисунок 1.6 - Ілюстрація архітектури серверного рендерингу (SSR) та взаємодії між сервером і клієнтом у веб-застосунку [15]

Таким чином, саме метафреймворки сформували сучасний підхід до розробки універсальних веб-застосунків, де клієнтська частина і серверні можливості працюють узгоджено в межах одного інструменту.

В категорії метафреймворків першим з’явився Next.js. Це метарамка на базі React, яка першою запровадила серверний рендеринг, гідрацію, автоматичне розбиття коду, файлову маршрутизацію та готову інфраструктуру для API-роутів. Того ж року з’явився й Nuxt.js, аналогічне рішення на базі Vue. Тут також було запезпечено SSR, статичну генерацію, конфігурацію за принципом “minimum setup”, модульність та єдину структуру проєкту. Окремо варто згадати й AnalogJS – метафреймворк для Angular, створений за аналогією з Next та Nuxt. Він надає Angular-проєктам підтримку SSR, файлової маршрутизації, статичної генерації сторінок та оптимізованого рендерингу, що традиційно виходило за межі стандартний можливостей Angular. У 2022 році з’явився Nuxt 3, повністю перероблена версія на Nitro-ядрі, що підтримує рендеринг у будь-яких середовищах (Node, serverless, edge), типізацію, покращену продуктивність і сучасний підхід до розробки.

Підсумовуючи, можемо впевнено сказати, що еволюція веб-фреймворків відображає тенденцію до підвищення інтерактивності, структурованості та продуктивності веб-застосунків. Якщо перше покоління інструментів вирішувало окремі технічні задачі, то сучасні фреймворки та метарамки створені на їхній базі, забезпечують повний цикл розробки, підтримуючи SSR, маршрутизацію, оптимізацію, роботу з API та модульність без необхідності ручного налаштування.

1.3. Порівняння сучасних версій трьох найпоширеніших веб-фреймворків

Огляд літератури, наукових досліджень та інтернет-ресурсів показав, що Angular, React та Vue залишаються трьома ключовими інструментами для розробки клієнтської частини веб-проєктів. Незважаючи на спільну компонентну модель, сучасні версії фреймворків суттєво відрізняються підходами до рендерингу, реактивності, управління станом, масштабованості та архітектури. Порівняння цих фреймворків дозволяє визначити їх сильні сторони та оптимальні сфери застосування.

Angular (сучасні версії, починаючи з Angular 2) є повноцінним фреймворком, який включає цілісну архітектуру та великий набір вбудованих інструментів. Він пропонує модульну структуру, шаблонний синтаксис, двосторонню прив'язку даних, засоби для роботи з формами, маршрутизацією, сервісами та залежностями.

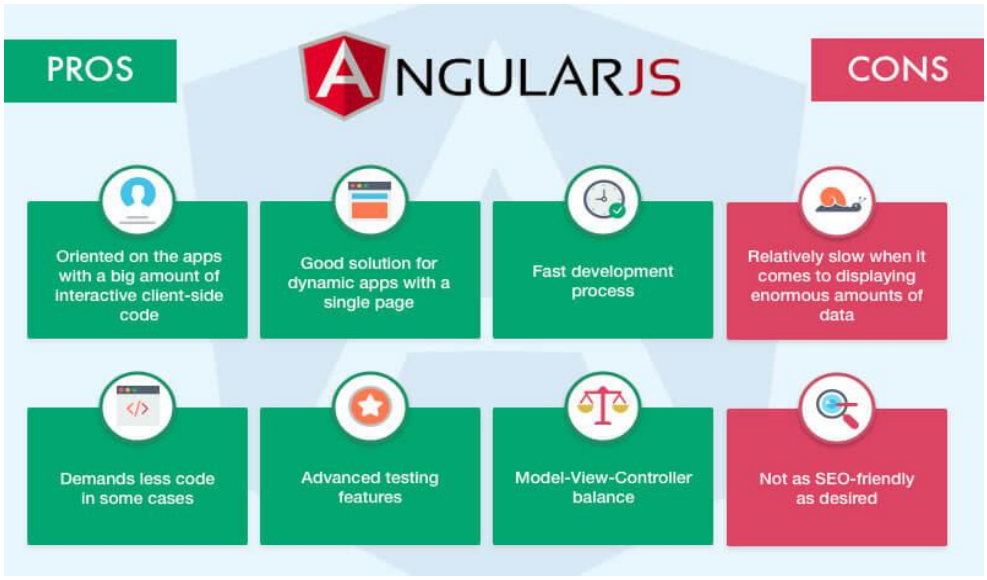

Рисунок 1.7 – Основні переваги та недоліки фреймворку Angular [16, 19]

Angular орієнтований на розробку комплексних застосунків і забезпечує високу ступінь структурованості коду. До базових можливостей Angular можна віднести:

- Повну платформу для створення великих систем;
- Чітку, жорстко визначену архітектура;
- Використання TypeScript як основної мови;
- Двосторонню прив’язку даних у шаблонах;
- Вбудовану маршрутизацію, наявність форм та систем керування залежностями;
- Оптимальність вибору для корпоративних проєктів.

React (найпоширеніші сучасні версії 18 та 19) позиціонується як бібліотека для побудови інтерфейсів, а не як повноцінний фреймворк. Його ключовою особливістю є використання віртуального DOM та декларативного підходу до опису інтерфейсу. React надає базовий каркас для створення компонентів, тоді як рішення щодо маршрутизації, керування станом і роботи з API приймаються на рівні проєкту. Базовими можливостями React є: компонентний підхід із декларативним рендерингом; використання віртуального DOM для оновлення інтерфейсу; гнучкість у виборі архітектури та додаткових бібліотек; проста інтеграції в існуючі проєкти; велика екосистема сторонніх інструментів; використання як у малих, так і у великих веб-застосунках.

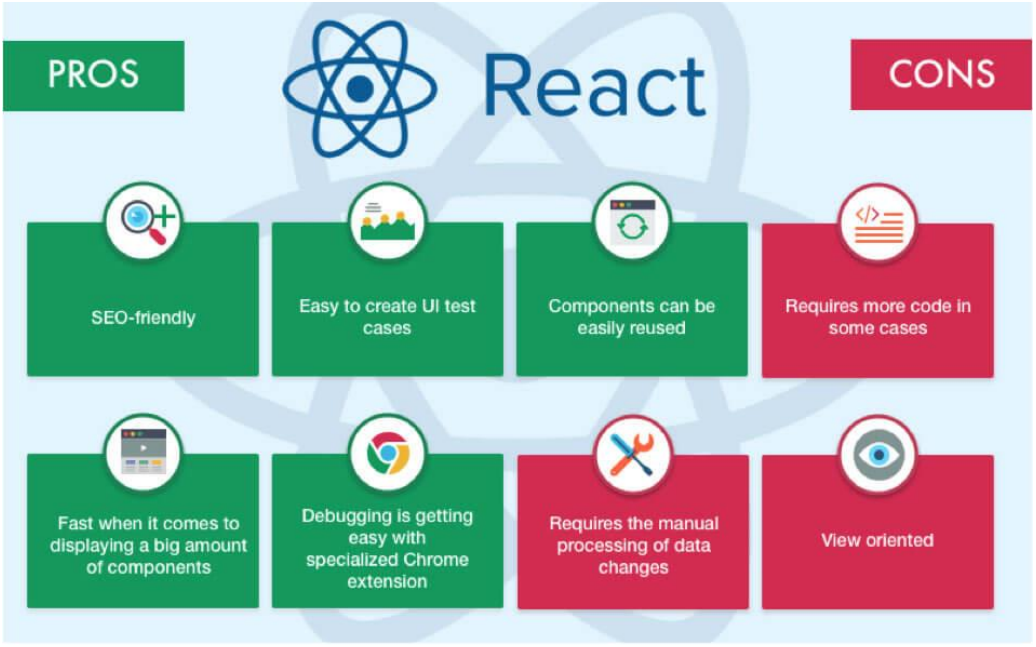

Рисунок 1.8 – Основні переваги й недоліки бібліотеки React [17, 19]

Vue 3 є прогресивним фреймворком, який поєднує простоту синтаксису з сучасною реактивною системою. Він використовує декларативні шаблони та компонентну модель, а також надає зручні інструменти для організації логіки всередині застосунку. Vue 3 зберігає низький поріг входу, залишаючись при цьому придатним для розробки складних та масштабованих систем.

Рисунок 1.9 – Основні переваги й недоліки бібліотеки Vue [19]

Третя версія фреймворку суттєво розширила можливості попередніх ітерацій за рахунок упровадження Composition API, що дозволило більш ефективно організовувати логіку компонентів, підвищувати повторне використання коду та покращувати масштабованість клієнтських застосунків. API композиції фактично є функцією, представленою у Vue 3, яка дозволяє розробникам організовувати код за логічними аспектами, а не за параметрами компонентів, такими як data, methods та computed [18]. Разом із цим фреймворк Vue 3 зберіг підтримку класичного Options API, що забезпечує зворотну сумісність і сприяє поступовому переходу на нові підходи.

Важливою перевагою Vue 3 є низький поріг входу, що робить його привабливим для початківців, а також достатня потужність для розробки складних, високонавантажених і масштабованих веб-застосунків. Фреймворк

забезпечує збалансоване поєднання декларативності, продуктивності та гнучкості, що позитивно впливає на ефективність процесу розробки.

На основі проведеного дослідження можна виділити такі базові можливості та переваги використання фреймворку Vue 3:

- застосування декларативних шаблонів і чітко структурованої компонентної архітектури веб-застосунку;
- вбудована реактивна система на рівні ядра фреймворку;
- підтримка сучасних підходів до організації логіки через Composition API;
- наявність офіційних інструментів екосистеми, зокрема Vue Router для маршрутизації та Pinia для керування станом застосунку;
- компактний, зрозумілий і добре документований програмний інтерфейс (API);
- низький поріг входу при збереженні можливостей для масштабування та підтримки складних проєктів.

У контексті порівняння сучасних клієнтських фреймворків Vue 3 виступає як компромісне рішення між простотою використання та функціональною насиченістю, що робить його конкурентоспроможним інструментом для створення клієнтської частини сучасних веб-застосунків (таблиця 1.3).

Таблиця 1.3 – Порівняльна таблиця базових опцій фреймворків

Критерій	Angular2+	React 18+	Vue 3
1	2	3	4
Тип інструменту	Повноцінний фреймворк	UI-бібліотека	Прогресивний фреймворк
Архітектура	Строга, модульна, DI	Відкрита, формується проєктом	Гнучка, компонентна
Основна мова	TypeScript	JavaScript / TypeScript	JavaScript / TypeScript
Реактивність / Оновлення UI	Двостороння прив’язка даних, структурні директиви	Віртуальний DOM, односпрямований потік даних	Реактивність через Proxy, гнучкі шаблони

Продовження табл.1.3

1	2	3	4
Організація проєкту	Висока структурованість, модулі, сервіси, шаблони	Мінімальна структура, залежність від сторонніх бібліотек	Офіційні інструменти, (Pinia, Vue Router), простий компонентний підхід
Маршрутизація	Вбудована	Через сторонні пакети (React Router)	Офіційний Vue Router
Керування станом	NgRx, або сервіси	Redux, Zustand та інші	Pinia (офіційно), VueX (legacy)
Поріг входу	Високий	Середній	Найнижчий
Продуктивність	Висока, оптимізація під час компіляції	Висока, ефективна робота через Virtual DOM	Висока, точкові оновлення завдяки реактивності
Масштабованість	Найкраще підходить для великих корпоративних систем	Підходить для будь-якого масштабу	Переважно малі та середні, але можлива масштабність
Гнучкість	Найменша, жорсткі правила	Найбільша	Середня, баланс гнучкості та структури
Екосистема	Повний набір інструментів	Найбільша кількість сторонніх рішень	Компактна офіційна екосистема
Сфера застосування	Великі SPA, корпоративні системи	Універсальні UI, SaaS, мобільні застосунки	Легкі інтерфейси, сайти, SPA середнього розміру

На основі таблиці 1.3 можна зробити висновок, що фреймворк Angular оптимальний для великих проєктів із чіткою структурою, React забезпечує найбільшу гнучкість і можливість адаптації під потреби команди, а Vue 3 поєднує простоту та ефективну реактивність, що робить його зручним у широкому спектрі завдань.

РОЗДІЛ 2.
ОБҐРУНТУВАННЯ, ВИБІР ТА РЕАЛІЗАЦІЯ ІНСТРУМЕНТАРІЮ ВИРІШЕННЯ ЗАДАЧІ

2.1 Обґрунтування вибору архітектури веб-застосунку

Архітектура веб-застосунку визначає спосіб, у який система опрацьовує дані, формує інтерфейс та взаємодіє з користувачем. Вибір архітектури має ключове значення для продуктивності, масштабованості, швидкості завантаження сторінок, SEO-оптимізації та зручності подальшої підтримки проєкту. Оскільки в рамках кваліфікаційної роботи для розробки клієнтського інтерфейсу буде застосовуватися технологічний стек Vue 3 + Nuxt 3 + Strapi, архітектурний вибір зводиться до тих моделей рендерингу та способів організації клієнт-серверної взаємодії, які підтримує Nuxt 3 як сучасний метафреймворк для Vue.

Незалежно від поєднання фреймворків процес рендиренгу складається із двох важливих частин, а саме самого рендиренгу та гідратації (рис.2.1).

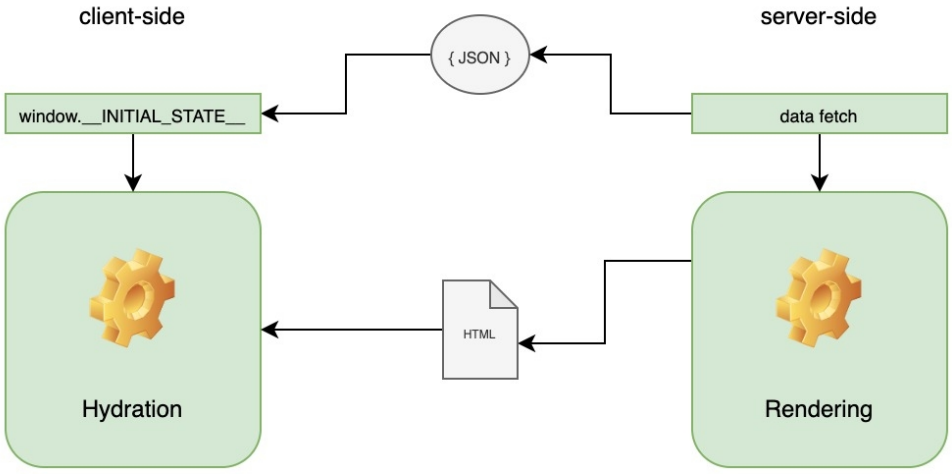

Рисунок 2.1 – Рендиренг на стороні сервера [19]

Гідратація клієнтської частини веб-застосунку у поєднанні з рендиренгом має дві основні цілі. По-перше, вона забезпечує зв’язування вже

згенерованої на сервері HTML-розмітки з клієнтським фреймворком, що передбачає ініціалізацію компонентів, реєстрацію обробників подій, форм і внутрішніх механізмів керування станом. По-друге, гідратація полягає у відновленні (реплікації) стану даних на стороні клієнта та призначенні відповідних значень властивостям і змінним застосунку.

Реалізація першої мети, як правило, здійснюється фреймворком автоматично в процесі виконання клієнтського коду, що дозволяє повторно використати вже відрендерену HTML-структуру без її повного перегенерування. Натомість друга складова потребує окремої уваги з боку розробника, оскільки коректність гідратації напряму залежить від узгодженості даних між серверною та клієнтською частинами.

На стороні сервера фреймворк формує початковий стан застосунку та присвоює дані відповідним змінним під час серверного рендерингу. Однак після передачі результату рендерингу до браузера ці дані необхідно повторно ініціалізувати на клієнті. Єдиними джерелами інформації в цьому випадку є HTML-розмітка, що вже відображена в браузері, а також початковий стан, який зазвичай передається через глобальний об’єкт window або вбудовується у сторінку у вигляді спеціального скрипту.

Таким чином, коректна реалізація гідратації забезпечує узгодженість стану між сервером і клієнтом, запобігає повторному рендерингу інтерфейсу та сприяє підвищенню продуктивності і стабільності веб-застосунку, особливо в архітектурах із серверним рендерингом або метафреймворках.

Nuxt 3 пропонує чотири основні архітектурні моделі для рендерингу сторінок, а саме CRS (Client-Side Rendering), SSR (Server-Side Rendering), SSG (Static Site Generation), Hybrid Rendering – гібридний режим, що поєднує попередні підходи.

Саме між цими варіантами здійснювався вибір у процесі проєктування. Нижче наведено аналіз кожної моделі в контексті вимог системи та особливостей застосування Nuxt 3.

Підхід SSG забезпечує високі показники продуктивності, подібні до серверного рендерингу (Server-Side Rendering, SSR), зокрема завдяки миттєвому відображенню контенту та мінімальному часу відповіді. Попередньо згенеровані сторінки можуть ефективно кешуватися та розповсюджуватися через мережі доставки контенту (CDN), що додатково підвищує швидкість доступу та стабільність роботи застосунку.

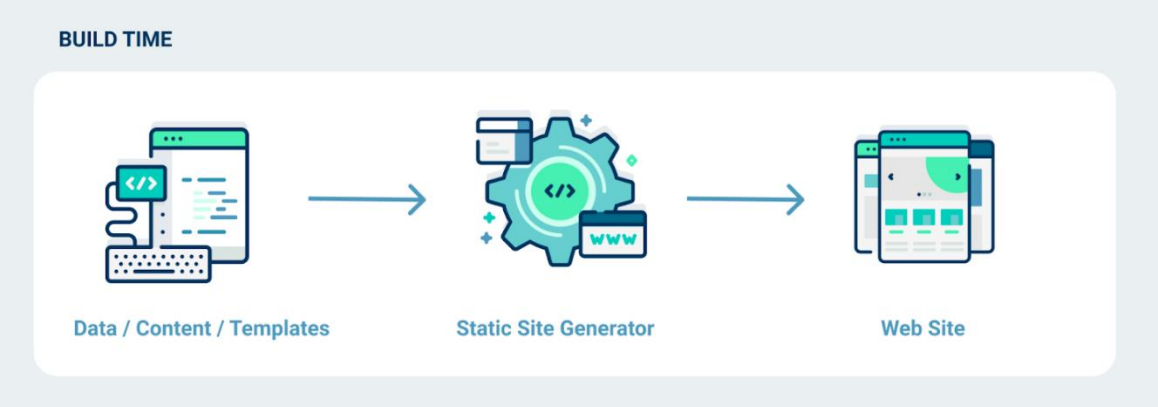

Рисунок 2.2. – Процес генерації статичних сайтів [20]

Генерація статичних сайтів є більш економічно доцільною та простішою в розгортанні порівняно з SSR-архітектурами, оскільки усуває потребу у постійній обробці HTTP-запитів на сервері. Це знижує навантаження на інфраструктуру, спрощує масштабування та зменшує витрати на хостинг. Водночас SSG особливо ефективний для веб-ресурсів зі здебільшого статичним або рідко змінюваним контентом. SSG не може бути використаний для сторінок з авторизацією чи персоналізацією і відповідно є не надто відповідним варіантом для отримуваного контенту в режимі реального часу.

Nuxt також підтримує тип архітектури SSG, який являє собою статичну генерацію сторінок під час білду. У цьому режимі система створює HTML-файли наперед, що дозволяє значно прискорити завантаження та розміщувати сторінки на CDN без потреба в рендерингу на сервері. До переваг SSG можна віднести наступні пункти: максимальна продуктивність, високу безпеку, низьку вартість інфраструктури. У проєкті за темою кваліфікаційної роботи

SSG може бути застосований лише для обмежених розділів, але не як основна архітектурна модель.

Client-Side Rendering у Nuxt передбачає, що формування інтерфейсу повністю виконується на клієнті. Сервер віддає мінімальний HTML-файл, після чого завантажується JavaScript, який відповідає за створення компонентів, маршрутизацію та створення сторінок. Такий підхід може забезпечувати високу інтерактивність після першого завантаження, мінімальне навантаження на сервер та підходить для сторінок, які не потребують SEO-індексації (наприклад, кабінет користувача). Проте CSR містить й ряд недоліків, зокрема тривалий час початкового завантаження; SEO-обмеження на сторінках, які все ж потребують індексації; залежить від продуктивності клієнтського пристрою. CSR не є оптимальним вибором для публічних сторінок, які повинні швидко відображатись й бути доступними для індексації пошуковими системами.

Server-Side Rendering в свою чергу дозволяє виконувати рендеринг компонентів на сервері. Готовий HTML надсилається користувачу, який одразу бачить вміст. Після цього Nuxt виконує гідрацію – активує клієнтський JavaScript, забезпечуючи інтерактивність.

Загалом, ми можемо виділити наступні плюси при використанні SSR: швидкість першого відображення (First Paint); повноцінна SEO-оптимізація; ефективна робота на слабких пристроях; можливість обробки даних на сервері (catching, auth, API middleware). Також можемо виділити додаткові переваги для застосунків та проєктів, де взаємодія з динамічним контентом відбувається через headless backend, в нашому випадку це Strapi. До цих переваг можна віднести виконання попередньої обробки запитів на отримання контенту, можливість інтеграції з системним кешуванням. При використанні даного підходу чутливі дані не потрапляютимуть у клієнтський код та забезпечуватиметься підвищена безпека завдяки серверним middleware.

Недоліком використання SSR є збільшення навантаження на сервер, складніша архітектура в порівнянні з CSR та вища її вартість.

SSR дозволяє нам реалізувати значну частину функціоналу серверної частини веб-застосунку без глибоких знань backend-розробки, при цьому використовуючи фреймворк для frontend-розробки.

Сучасною концепцією веб-розробки, спрямованою на подолання обмежень як серверного рендерингу (Server-Side Rendering, SSR), так і генерації статичних сайтів (Static Site Generation, SSG) є інкрементальна статична регенерація (Incremental Static Regeneration, ISR).

Цей тип концепції поєднує переваги обох підходів, забезпечуючи високу продуктивність статичних сторінок разом із можливістю їх динамічного оновлення без необхідності повної повторної збірки всього веб-застосунку (рис. 2.3).

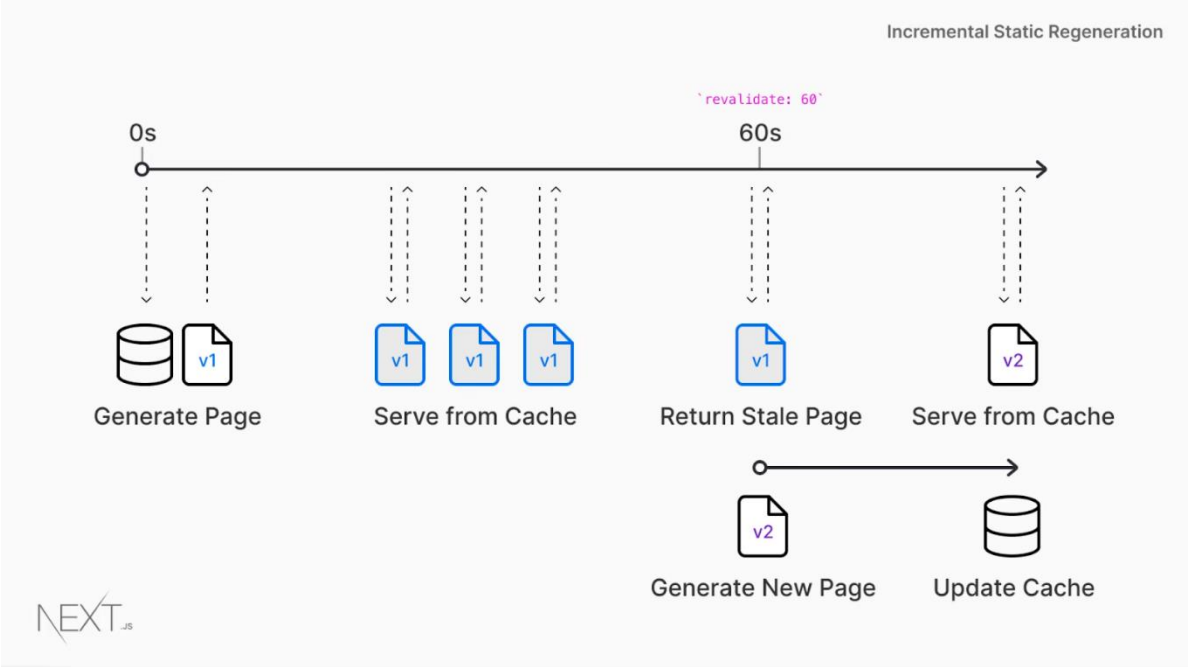

Рисунок 2.3 – Опис процесу інкрементальної статична регенерації [20]

У Nuxt 3 інкрементальна статична рененерація реалізується не як окремий модель, а через механізм Hybrid Rendering та route rules. Фреймворк дозволяє поєднувати статичну генерацію з можливістю перегенерації окремих сторінок після їх публікації, що концептуально відповідає підходу ISR.

Гібридність – це одна з основних переваг Nuxt 3. Розробник може визначати режим рендерингу окремо для кожної сторінки. Залежно від типу конкретної сторінки він індивідуально обирає який тип використати. Для прикладу публічні сторінки будуть використовувати SSR, адмін-панелі будуть відображатись за допомогою CSR, повністю статичний контент може бути показаний за допомогою SSG. Гібридна архітектура має значні переваги над іншими, зокрема наявністю можливості оптимізувати кожну сторінку окремо. Розробник може розраховувати на економію ресурсів сервера там, де немає необхідності рендерити контент безпосередньо на ньому, підвищення швидкодії та реалізацію природного масштабування. Даний тип архітектури може бути універсальним до будь-якого типу контенту, навіть в межах одного веб-застосунку. Саме гібридність робить Nuxt 3 оптимальним рішенням для сучасних багатофункціональних застосунків.

Проаналізувавши вимоги до системи, ми виділили наступні критерії, на основі яких доцільно обирати тип архітектури веб-застосунку:

1) Швидкість початкового завантаження сторінок;
2) Оптимізація для SEO;
3) Динамічність отримання даних зі Strapi;
4) Стабільність та передбачуваність поведінки;
5) Зручність масштабування та підтримки застосунку в майбутньому;
6) Можливість реалізації кешування;
7) Безпечність взаємодії з API;
8) Забезпечення оптимальної роботи різних пристроях.

З цього випливає, що CRS не підходить для проекту кваліфікаційної роботи через SEO-вимоги, адже сайт повинен гарно індексуватись та взаємодіяти з пошуковими системами. SSG не покриває динамічна сценарії, а використання виключно SSR створює зайве навантаження на сервер. Єдиним раціональним вибором для веб-застосунку є гібридна SSR-архітектура (таблиця 2.1).

Таблиця 2.1 – Переваги гібридної SSR-архітектури в Nuxt 3

Перевага	Короткий зміст та технічне обґрунтування
Миттєве відображення сторінок та покращений UX	HTML формується на сервері, тому користувач одразу бачить готовий контент без затримок, незалежно від швидкості пристрою
Повноцінна SEO-індексація	Пошукові системи отримують повністю відрендерений HTML, що підвищує видимість ресурсу та рейтинг у видачі
Безпечна взаємодія зі Strapi	Запити до Strapi проходять через Nuxt server-side middleware, що приховує токени, обробляє помилки та не передає чутливі дані на клієнт.
Оптимізація продуктивності через кешування	Nuxt може кешувати серверні відповіді, зменшуючи навантаження на Strapi та пришвидчуючи повторні запити
Відсутність дублювання логіки	У SSR логіка роботи з даними виконується на сервері один раз, а клієнт отримує вже готовий результат, що спрощує структуру застосунку
Гнучке застосування різних режимів рендерингу	Nuxt дозволяє використовувати CSR, SSG, SSR для кожного маршруту окремо, що дозволяє використовувати ресурс пристрою та сервера найоптимальнішим чином

Отже, в ході виконання кваліфікаційної роботи вдалось визначитись з типом архітектури для нашого веб-застосунку й обрати найоптимальніший варіант, який задовольняє абсолютно всі потреби і вимоги.

2.2. Обґрунтування вибору стеку технологій для реалізації веб-застосунку

Вибір технологічного стеку є ключовим етапом проєктування веб-застосунку, оскільки саме від нього залежить швидкість розробки, масштабованість, продуктивність, гнучкість архітектури та можливість подальшої підтримки системи. У рамках цього проєкту було прийнято рішення використовувати стек Vue + Nuxt 3 + Strapi, який поєднує у собі сучасний реактивний фреймворк, потужний метафреймворк для серверного рендерингу та headless-платформу для керування даними. Такий вибір є

результатом аналізу вимог застосунку, співставлення доступних технологій та оцінки їхнього потенціалу у контексті розробки багатомовного, SEO-оптимізованого та динамічного веб-ресурсу.

Вибір основного інструменту для розробки клієнтської частини впав саме на Vue, адже цей фреймворк належить до сучасних JavaScript-фреймворків, орієнтованих на створення реактивних інтерфейсів та компонування застосунків. Його ключовими перевагами є простота синтаксису, низький поріг входу, висока продуктивність та гнучка архітектура, яка дозволяє масштабувати застосунок без втрати зрозумілості структури.

Таблиця 2.2 – Основні характеристики Vue що аргументують вибір фреймворку для використання в даному проєкті

Тип характеристики	Опис
Компонентна модель	Vue забезпечує чітке структурування інтерфейсу на окремі багаторазові компоненти, що полегшує розробку та супровід проєкту. Це особливо важливо у застосунку, який містить велику кількість ізольованих UI-елементів – фільтри, картки, банери, списки та інші модулі
Сучасна реактивність	Завдяки Composition API у Vue 3 забезпечено прозору роботу з реактивними даними, підвищену продуктивність та покращений контроль над логікою компонентів. Це впливає на ефективність оновлення інтерфейсу та стабільність роботи в динамічних секціях застосунку
Прозорість та передбачувність поведінки коду	Vue забезпечує інтуїтивну логіку компонування, що зменшує кількість помилок та пришвидшує процес розробки. Для дипломного проєкту це особливо цінно, адже дозволяє сконцентруватися на функціональності, а не на боротьбі з інструментом
Широка екосистема та розвинена інфраструктура	Екосистема Vue включає бібліотеки для роботи з формами, валідацією, рендерингом, станом, мультимовністю. Це дозволяє зменшити час на розробку й уникнути дублювання типових рішень

Усі наведені в таблиці 2.2. чинники роблять Vue оптимальним вибром для швидкої та гнучкої реалізації інтерфейсної частини застосунку.

Що ж стосується Nuxt 3, то він є сучасним метафреймворком над Vue, який спрощує створення продуктивних веб-застосунків, забезпечує SSR, гібридний рендеринг, модульність та готову структуру проєкту.

Таблиця 2.3 – Основні характеристики Nuxt 3, що обґрунтовують ефективність використання фреймворку

Характеристики	Обґрунтування
Підтримка SSR та гібридного рендерингу	Nuxt надає можливість обирати режим рендерингу для кожної сторінки окремо, SSR для публічних сторінок, SSG для статичних розділів та CSR для приватних панелей або інтерактивних модулів. Такий підхід забезпечує максимальну продуктивність та ефективне використання ресурсів сервера
Покращена SEO-оптимізація	На відміну від класичних SPA, Nuxt дозволяє формувати повноцінний HTML на сервері, що забезпечує коректну індексацію контенту. Це критично для багатомовного застосунку, де кожна локалізована сторінка повинна бути окремо доступною для пошукових систем
Модульна система та автоматична конфігурація	Nuxt має великий каталог модулів, багато з яких використовується у проєкті, наприклад: - @nuxtjs/i18n для мультимовності, - @nuxtjs/axios для HTTP запитів, - @pinia/nuxt для керування станом, - @vee-validate/nuxt для роботи з валідацією форм та полів, - @nuxt/image для оптимізації зображень. Це скорочує кількість ручних налаштувань і пришвидчує розробку
Server Routes і middleware	Nuxt 3 дозволяє створювати серверні маршрути, які виступають проміжним шаром між клієнтом і бекендом. У рамках цього проєкту це забезпечило: - Захист чутливих даних при зверненні до Strapi; - Можливість кешування на рівні сервера; - Централізовану обробку помилок API. Ці можливості є важливими для побудови стабільної й масштабованої системи
Оптимізована структура проєкту	Файлова маршрутизація, автоматичне підключення компонентів та конфігурація за принципом “minimum setup” забезпечили високу швидкість розробки та зменшили кількість шаблонного коду.

Використання метафреймворку разом з основним фреймворком дозволяє значно спростити та пришвидшити процес розробки, розширивши функціонал веб-застосунку. Описані в таблиці 2.3 характеристики Nuxt 3 аргументують вибір зроблений на користь використання даного фреймворку для розробки веб-застосунку, адже цей інструмент дозволив мені швидко та якісно писати масштабований код.

Strapi – це headless CMS, побудована на Node.js, яка дозволяє створювати кастомні моделі даних і автоматично генерувати REST або GraphQL API.

Таблиця 2.4 – Переваги використання CMS Strapi для менеджменту контенту

Критерій	Обгрунтування
Чітке розділення фронтенду та бекенду	Strapi працює незалежно від Nuxt, надаючи API-інтерфейс для отримання контенту. Це сприяє модульності системи, можливості масштабування та повторного використання контенту в інших застосунках
Панель керування контентом	Створення та редагування контенту може виконуватися без залучення розробників, що важливо для реальних веб-проєктів, орієнтованих на часте оновлення інформації
Гнучкість у моделюванні даних	Strapi дозволяє створювати будь-які сутності з потрібними полями, відношеннями та типами, що забезпечує відповідність структурі фронтенд-компонентів у Nuxt
Масштабованість та незалежність від платформи	Strapi легко переноситься між середовищами, підтримує Docker, а також дозволяє реалізувати авторизацію, контроль доступу та інші механізми бекенду без створення всього API вручну

У межах даного проєкту Stapi було обрано через ряд переваг, які дозволили обійтись без написання повноцінної серверної частини для нашого веб-застосунку.

У процесі аналізу також розглядались інші рішення (табл. 2.5).

Таблиця 2.5 – Порівняння вибраного стеку з альтернативами

Альтернатива	Причина відмови
React + Next.js	Вища складність структури, потреба у більшій кількості сторонніх бібліотек для задач, які у Nuxt вирішуються нативно
Angular	Надмірна складність для проєктів із високими вимогами до гнучкості, складніша крива навчання
WordPress / ACF	Низька продуктивність, складність масштабування, обмеження архітектури, слабка підтримка SSR, неможливість повноцінно реалізувати весь необхідний функціонал

В ході виконання дослідження було здійснено вибір стеку технологій в комбінації Vue 3, Nuxt 3 та Strapi, що є обґрунтованим рішенням, яке забезпечуватиме: гнучку та масштабовану архітектуру; продуктивний SSR та гібридний рендеринг, зручну роботу з динамічним контентом; багатомовність на рівні архітектури; високий рівень SEO-оптимізації; стабільний і безпечний серверний шар; швидкість розробки, якість та чисту структуру коду. Цей стек технологій повністю відповідає вимогам проєкту, дозволяючи реалізувати сучасний, динамічний, багатомовний та високопродуктивний веб-застосунок.

2.3 Перспективи використання технологічного стеку для оптимізації та розвитку веб-застосунку

Вибраний стек технологій – Vue , Nuxt 3 та Strapi – не тільки задовольняє поточні вимоги проєкту, але й забезпечує значний потенціал для подальшого розвитку системи. Перспективність використання цього стеку визначається масштабованістю архітектури, активним розвитком відповідних інструментів, можливістю інтеграції нових функціональних модулів та стабільністю екосистеми. У цьому підрозділі розглянуто напрямки та переваги майбутнього розширення і вдосконалення застосунку в межах обраної технологічної платформи.

Vue та Nuxt пропонують компонентний підхід до розробки, що дозволяє безболісно нарощувати кількість UI-модулів та логічних блоків незалежно від їхньої складності. Компоненти залишаються ізольованими та автономними, а завдяки Composition API можна формувати повторно використовувані логічні частини (composables), що значно спрощує додавання нових функцій. Таке розділення дає змогу легко додавати нові сторінки та розділи без перегляду всієї структури застосунку, повторно використовувати логіку API-запитів, валідації та реактивних моделей, а також масштабувати застосунок горизонтально, розширюючи команду розробників без виникнення конфліктів у коді. Надалі це дозволяє реалізовувати нові модулі – наприклад, персоналізовані профілі, систему реєстрації, розширені фільтри або інтерактивну аналітику – без змін основ архітектури.

Окрім цього, екосистема Vue активно розвивається, виходять нові інструменти для тестування, керування станом, анімаціями. З’являються нові оптимізації та покращення. Nuxt, у свою чергу, впроваджує сучасні технології (ESM, Vite, server islands, edge-rendering), що робить його конкурентним у порівнянні зі світовими метафреймворками. Ці всі тенденції гарантують, що застосунок можна буде безболісно оновлювати, трансформувати відповідно до нових стандартів, інтегрувати в нього сторонні сервіси та API, а також підтримувати актуальність використаних технологій протягом тривалого часу.

Strapi в свою чергу також не стоїть на місці, даний інструмент, як headless CMS забезпечує еластичну роботу з контентом та можливість моделювання складних структур даних. Його розвиток у напрямку plug-ins, авторизаційних модулів, GraphQL-підтримки та cloud-рішень відкриває перспективи до підключення зовнішніх баз даних, створення автоматизованих API для мобільних застосунків, впровадження кастомної логіки та масштабування контенту без зміни фронтенду. Таким чином Strapi дає змогу розвивати застосунок не лише як сайт, а й як частину більшої цифрової платформи.

Перспективи використання стеку Vue 3 + Nuxt 3 + Strapi є широкими та довгостроковими. Комбінація сучасного реактивного фреймворку, потужного SSR-метафреймворку та гнучкої headless CMS забезпечує можливість розгортання, підтримки і масштабування складних веб-застосунків. Такий стек дозволяє адаптувати систему до зростаючих потреб користувачів, розширювати функціональність та підтримувати високий рівень продуктивності, що підтверджує його актуальність у перспективі кількох років розвитку.

РОЗДІЛ 3.
РЕЗУЛЬТАТИ ВИРІШЕННЯ ЗАДАЧІ

3.1. Реалізація інтерфейсної частини веб-застосунку на базі Vue 3 та оцінка її ефективності

Архітектура клієнтської частини розробленого веб-застосунку побудована за компонентним підходом, що дозволяє ізолювати логіку, повторно використовувати UI-елементи та оптимізувати процес розробки. У проєкті сформовано чотири основні групи компонентів: базові та функціональні UI-елементи, секційні компоненти сторінок та глобальні елементи макета. Такий поділ забезпечує структурованість і мінімізує дублювання коду, що є ключовим проявом ефективності використання сучасного стеку Vue 3 + Nuxt 3.

Розглянемо реалізацію UI-компонентів. До цієї групи належать універсальні елементи інтерфейсу, які застосовуються на всіх сторінках застосунку. Вони не містять бізнес-логіки та виконують роль будівельних блоків для складніших компонентів. Серед них: Notification.vue, Input.vue, Textarea.vue, Chip.vue, SkeletonItem.vue, Avatar.vue, Button.vue, Checkbox.vue та багато інших.

Для прикладу візьмемо компонент Notification.vue. Цей елемент відповідає за відображення повідомлень системи: інформаційних, успішних, попереджувальних та помилкових. Завдяки параметризованій структурі компонент підтримує різні стани без необхідності дублювання стилів — тип повідомлення визначається динамічно, також компонент має також автоматичне закриття через таймер, що підвищує зручність для користувача та звільняє розробника від ручної реалізації таких сценаріїв у кожному модулі. Код реалізації даного компонента можна переглянути в додатку А.

Також розглянемо компонент Chip.vue. Він використовується для відображення активних фільтрів у секції Residents. Компонент

мінімалістичний, складається з відображення значення та кнопки видалення. Така інкапсуляція дозволяє повторно використовувати Chip у будь-якому контексті, наприклад у фільтрах чи вибірках. При цьому логіка закриття повністю винесена в emit-подію, що забезпечує гнучкість та дозволяє керувати відображенням повністю з батьківського компонента. Код реалізації даного компонента можна переглянути в додатку Б.

UI-компоненти різної складності формують дизайн-систему застосунку, забезпечують консистентність і значно пришвидшують розробку та майбутнє масштабування. Що ж стосується секційних компонентів, то в цьому проєкті вони відіграють роль логічних блоків для сторінок, які відповідають за окремі частини бізнес-функціоналу. Враховуючи те, що головна сторінка містить в собі багато відокремлених секцій, для розширеного відображення яких згодом будуть реалізовуватись окремі сторінки, то було прийняте рішення розбити головну сторінку на дрібніші частини й винести всю логіку, зокрема логіку отримання даних, безпосередньо в секційні компоненти. Список реалізованих в проєкті секцій виглядає наступним чином:

- Community;
- Residents;
- Membership;
- Events;
- Partners;
- Contact.

Кожна з них може бути використана повторно в майбутньому при масштабуванні та реалізації нових сторінок у веб-застосунку. Окрім цього, при створенні даних модулів була реалізована структура, яка дозволяє без зайвих проблем розширювати функціонал та доповнювати секції новими елементами, не ускладнюючи при цьому загальну архітектуру (рис.3.1).

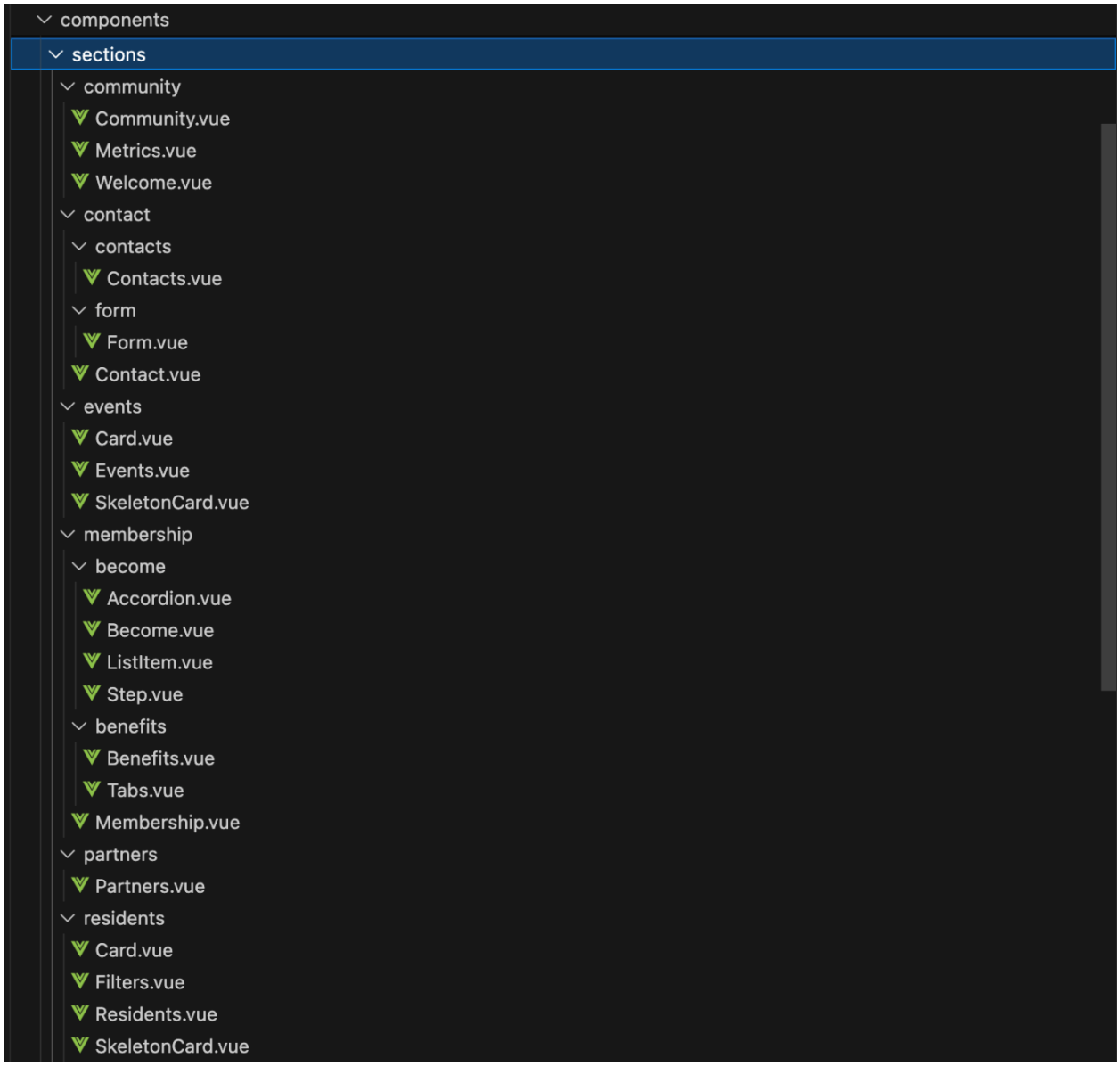

Рисунок 3.1 – Структура реалізації секційних та дочірніх для них компонентів

Для детальнішого огляду реалізації секційних модулів, пропоную розглянути приклад реалізації секції Residents, яка складається з наступних Vue компонентів: Residents.vue (додаток В), Filters.vue (додаток Г), Card.vue (додаток Ґ), SkeletonCard.vue (додаток Д).

Residents поєднує в собі отримання даних зі Strapi, реактивну роботу з фільтрами, механізм пагінації, використання різних бібліотек, роботу зі станом застосунку та багато іншого. В секції можна побачити різні варіанти отримання даних зі Strapi, як за допомогою виключно інструментів Vue 3 так і за допомогою інструментів Nuxt 3. Залежно від цілей встановлених бізнес-

логікою в ході виконання кваліфікаційної роботи ми звертались до різних способів реалізації даного функціоналу. Окрім цього, в коді можна побачити, як відбувається фільтрація та автоматична взаємодія між різними групами фільтрів, пагінація для отримання контенту порціями та його довантаження в разі потреби, використання строгої типізації для уникнення помилок, перевикористання UI-блоків, умовний рендеринг та багато чого іншого.

Все це відбувається в окремій структурній одиниці, яка є повністю незалежною від інших частин застосунку та дозволяє уникати впливу складної логіки в одній частині проєкту на логіку в інших його частинах, при цьому зберігаючи гнучкість та здатність легко масштабуватись без зміни логіки взаємодії.

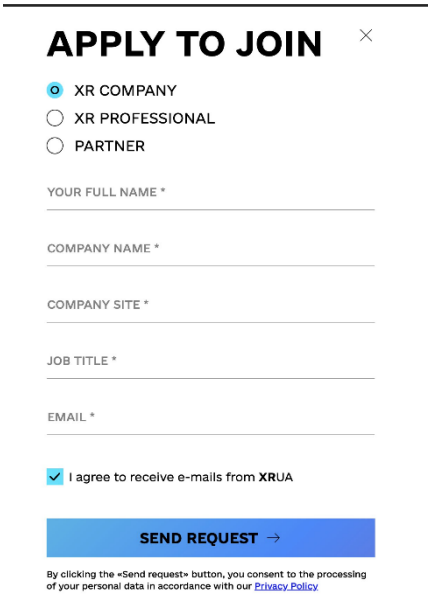

Рисунок 3.2. – Вигляд сторінки проекту з формою Apply-to-join

Окрім вищеперерахованих типів компонентів проєкт також містить ряд глобальних компонентів та елементів макетів сторінок. До них відносимо Header.vue, Footer.vue, Navigation.vue, MobileNavigation.vue, LanguageSwitcher.vue, Notifications.vue. Всі ці компоненти мають певний функціонал чи властивості, що мають змогу впливати на застосунок глобально, або ж відображати певні елементи, які не прив’язані до тієї чи іншої сторінки, або ж навпаки, повинні бути абсолютно всюди.

Яскраві приклади – це зміна мови на сайті, або ж нотифікації, які повинні мати змогу відобразитись на абсолютно кожній сторінці, підвал сайту, спільний для практично кожної сторінки, або ж меню навігації.

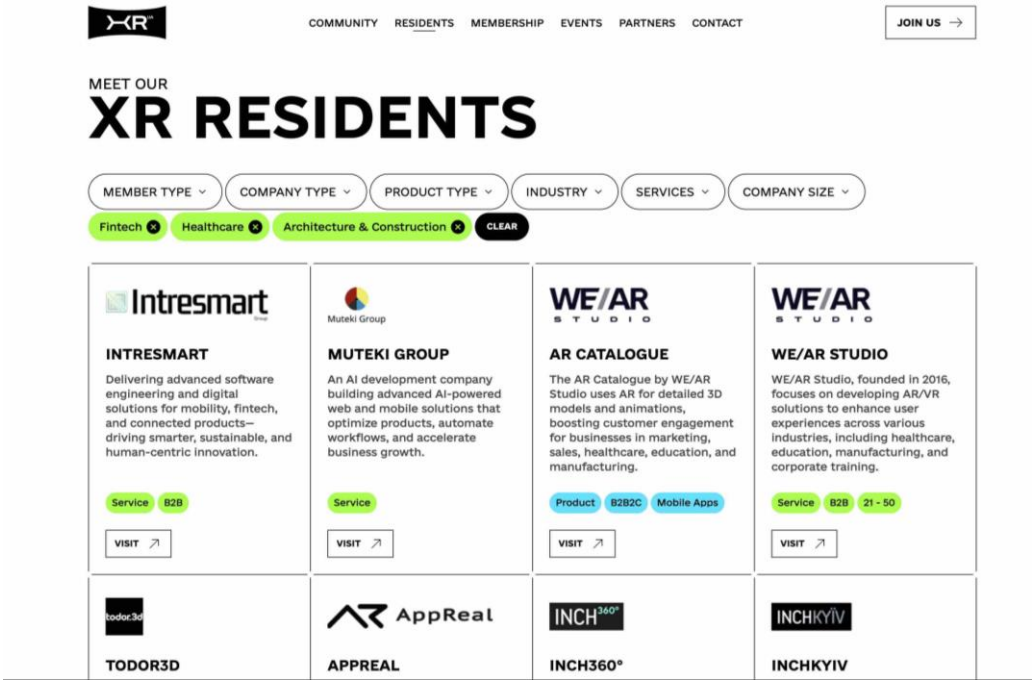

Рисунок 3.3. – Фрагмент сайту із списком резидентів та активними фільтрами

На рисунку 3.4 представлено приклад використання спеціального візуального елементу в інтерфейсі веб-застосунку, який дозволяє користувачу застосовувати фільтрацію на сайті (за параметрами, категоріями, пошуком тощо). Якщо результатів за заданими критеріями немає (0 записів), замість порожнього екрану відображається плейсхолдер у вигляді текстове повідомлення - підказки.

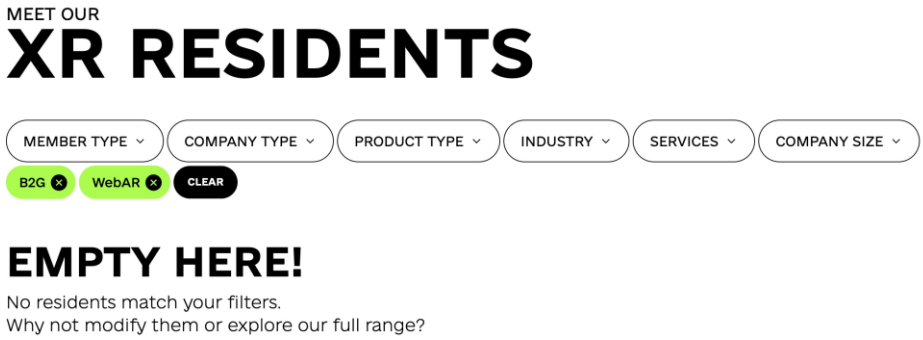

Рисунок 3.4. – Приклад відображення плейсхолдера на сайті

В підсумку, можемо зазначити, що сформована система компонентів мінімізує дублювання, пришвидшує розробку, спрощує підтримку та дозволяє розширювати застосунок без зміни його архітектури. Все це демонструє високу ефективність використання фреймворків в розробці веб-застосунків завдяки чіткому поділу відповідальності, гнучкості та повторному використанню компонентів, ізоляції бізнес-логіки, можливості масштабування тощо.

3.2. Реалізація архітектури та взаємодії API у Nuxt 3 із застосуванням Strapi

Архітектура застосунку побудована навколо зв'язки Nuxt 3 + Strapi, де Nuxt відповідає за клієнтську частину та проксі-доступ до API, а Strapi виконує роль headless CMS. Додатково інтегровано зовнішній сервіс Sendpulse для відправки листів із форми подачі заявки. Рішення сфокусоване на тому, щоб винести всю роботу з мережею в окремі шари, мінімізувати дублювання коду та забезпечити контроль над кешуванням і безпекою.

Усі запити до CMS інкапсульовано в окремих API-модулях у директорії api/strapi. Кожен модуль відповідає за окремий ендпоїнт чи групу ендпоїнтів, наприклад events.api.ts - за події, filters.api.ts - за фільтри, join-us-form.api.ts - за форму заявки, а support-us.api.ts - за отримання даних, що відображаються в модальном вікні з банківськими реквізитами. Приклад реалізації у файлі api/strapi/events.api.ts:

```
import { formatStrapiSortParameter } from '~/helpers';
import { createXrStrapiApi } from '.';
import type {
  IStrapiPagination,
  IEvent,
  IEventPopulate,
  IStrapiListResponse,
  IStrapiSort,
  IStrapiFilters,
} from '~/types';
```

```
export function useEventsApi() {
  const { xrStrapiApi } = createXrStrapiApi();

  function getEvents(
    populate: IEventPopulate = {},
    filters: IStrapiFilters = {},
    pagination: Partial<IStrapiPagination> = {},
    sortObj: IStrapiSort<IEvent> = {},
  ): Promise<IStrapiListResponse<IEvent>> {
    const sort = formatStrapiSortParameter(sortObj);

    return xrStrapiApi.get('/events', { params: { populate, filters, pagination, sort } })
      .then((res) => {
        return res.data;
      });
  }
  function getEvent(id: number | string, populate: IEventPopulate = {}): Promise<IEvent> {
    return xrStrapiApi.get('/events/' + id, { params: { populate } })
      .then((res) => {
        return res.data.data;
      });
  }
  return {
    getEvents,
    getEvent,
  }
}
```

```
import { formatStrapiSortParameter } from '~/helpers';
import { createXrStrapiApi } from '.';
import type {
  IStrapiPagination,
  IEvent,
  IEventPopulate,
  IStrapiListResponse,
  IStrapiSort,
  IStrapiFilters,
} from '~/types';

export function useEventsApi() {
  const { xrStrapiApi } = createXrStrapiApi();

  function getEvents(
    populate: IEventPopulate = {},
```

```
    filters: IStrapiFilters = {},
    pagination: Partial<IStrapiPagination> = {},
    sortObj: IStrapiSort<IEvent> = {},
  ): Promise<IStrapiListResponse<IEvent>> {
    const sort = formatStrapiSortParameter(sortObj);

    return xrStrapiApi.get('/events', { params: { populate, filters, pagination, sort } })
      .then((res) => {
        return res.data;
      });
  }
  function getEvent(id: number | string, populate: IEventPopulate = {}): Promise<IEvent> {
    return xrStrapiApi.get('/events/' + id, { params: { populate } })
      .then((res) => {
        return res.data.data;
      });
  }
  return {
    getEvents,
    getEvent,
  }
}
```

Нижче наведено приклад безпосереднього використання даного хуку в коді:

```
const populate: IEventPopulate = {
  industries: true,
  country: true,
  organizer_type: true,
  event_types: true,
  event_modes: true,
  label: true,
};
const filters = buildDateFilter();
const pagination: Partial<IStrapiPagination> = { pageSize: 4 };
const sort: IStrapiSort<IEvent> = { date_start: 'asc' };
const { data, refresh, error, status } = await useAsyncData(
  ASYNC_DATA_CASH_KEYS.EVENTS.MAIN_PAGE_LIST,
  () => getEvents(
    populate,
    filters,
    pagination,
    sort,
  ),
  {
```

```
  lazy: true,
  getCachedData: (key, app) => app.payload?.data?.[key] ?? app.static?.data?.[key],
  server: false,
 },
);
```

Усі ці модулі працюють через спільну фабрику createXrStrapiApi(), яка створює клієнт із правильною базовою адресою. Це дає можливість централізовано змінити URL або налаштування без переписування компонентів. Сама ж фабрика зберігається у api/strapi/index.ts та виглядає наступним чином:

```
import axios from 'axios';
let xrStrapiApiInstance: ReturnType<typeof axios.create> | null = null;
export function createXrStrapiApi() {
 const runtimeConfig = useRuntimeConfig();
 const baseUrlEndpoint = '/api/strapi'
 if (xrStrapiApiInstance) {
  return { xrStrapiApi: xrStrapiApiInstance };
 }
 xrStrapiApiInstance = axios.create({
  baseURL: import.meta.server ? runtimeConfig.public.baseUrl + baseUrlEndpoint :
baseUrlEndpoint,
  timeout: 10000,
 });
 xrStrapiApiInstance.interceptors.response.use(
  (response) => response,
  (error) => {
   console.log('Strapi Response Interceptor API Error', error.response?.data || error.message);
   return Promise.reject(error);
  }
 );
 return {
  xrStrapiApi: xrStrapiApiInstance,
 }
}
```

Такий підхід ефективний тим, що компоненти сторінок взагалі не знають про деталі REST-ендпоїнтів. Вони працюють із простими функціями на кшталт getEvents() або getSupportUsModalData(), що зменшує зв’язність та спрощує подальші зміни у схемі Strapi.

Окрім цього, доступ до зовнішнього Strapi організовано через єдиний серверний вхід server/api/strapi/[...path].ts. Будь-який запит з фронтенду летить не напряму до CMS, а на Nuxt-сервер, де обробляється утилітою proxyToStrapi(). В утиліті ми вирішуємо одразу декілька задач, а саме:

- Приховуємо від клієнта наші секретні токени доступу;
- Обробляємо помилки Strapi в єдиному місці;
- Маємо можливість додавати додаткові заголовки та логи.

Для GET-запитів поверх цього проксі використовується defineCachedEventHandler() з режимом swr і власним ключем кешу:

```
import { makeCacheKey, proxyToStrapi } from '~/server/utils/strapi';

export default defineCachedEventHandler(async (event) => {
  return proxyToStrapi(event);
}, {
  swr: true,
  staleMaxAge: 60 * 10,
  getKey: (event) => makeCacheKey(event),
});
```

Функція makeCacheKey() враховує path та query-рядок, тому кожна різна комбінація фільтрів або пагінації кешується окремо. Це знижує кількість запитів до Strapi при повторному використанні тих самих даних різними користувачами. Фактично Nuxt виступає у ролі проміжного кешуючого шару з політикою stale-while-revalidate, в якій користувачі швидко отримують закешовану відповідь, а в фоні може бути виконана перевірка актуальності даних.

Також в застосунку для більшості секцій використовується Nuxt-хук useAsyncData() з явним керуванням опціями lazy, server та getCachedData. Такий підхід має кілька характерних рис, а саме:

1) Завантаження переноситься на клієнт (через server: false), що дає контрольований, передбачуваний сценарій гідрації та можливість одразу відобразити сторінку зі скелетонами, аби користувач побачив певний контент ще до того, як дані будуть звантажені,

2) lazy: true дозволяє рендерити базову розмітку (ті ж скелетони, заголовки секцій тощо) без блокування на очікуванні відповіді сервера,

3) getCachedData використовує payload/статичні дані, якщо вони були збережені Nuxt, тому при повторному візиті не вимагають повторного звернення до API,

4) ключі кешу (ASYNC_DATA_CASH_KEYS) централізують управління даними.

Для Residents дані списку отримуються окремо від фільтрів, оскільки результати сильно залежать від інтерактивної взаємодії користувача. Фільтри кешуються через useAsyncData, а запит резидентів виконується окремо з урахуванням поточних фільтруючих параметрів. Зупинимось на цьому детальніше, адже це дуже яскравий приклад того, як Vue та Nuxt дозволяють поєднати складну бізнес-логіку з керованою структурою коду.

Компонент Filters.vue в секції працює на базі конфігураційного масиву filterConfig, що виглядає наступним чином:

```
const filterConfig: readonly IStrapiRelationFilterConfigItem[] = shallowReadonly([
  { getter: getMemberTypes,  selectedOptionsRef: memberTypeSelectedOptions,  strapiKey: 'member_type' },
  { getter: getCompanyTypes, selectedOptionsRef: companyTypeSelectedOptions, strapiKey: 'company_type' },
  { getter: getProductTypes, selectedOptionsRef: productTypeSelectedOptions, strapiKey: 'product_type' },
  { getter: getIndustries,   selectedOptionsRef: industrySelectedOptions,   strapiKey: 'industries' },
  { getter: getServices,     selectedOptionsRef: serviceSelectedOptions,    strapiKey: 'services' },
  { getter: getCompanySizes, selectedOptionsRef: companySizeSelectedOptions,strapiKey: 'company_size' },
]);
```

Рисунок 3.5 – Конфігураційний масив filterConfig

Завдяки такій реалізації асинхронне завантаження опцій усіх фільтрів реалізовано однією функцією getDropdownsOptions(), де масив filtersGetters проходить через Promise.all. Логіка побудови об’єкта фільтрів також універсальна – для кожного елемента конфігурації генерується вкладена структура slug: { $in: […] }. Додавання нових фільтрів, (наприклад за локацією), зводиться до одного запису у filterConfig та додаткового getter’а в useFiltersApi().

Окремий watcher контролює взаємовиключні групи фільтрів (варіанти member type product/service), і вмикає або вимикає пов’язані селектори, що дозволяє уникати некоректних комбінацій при побудові запитів. Також

сформовані фільтри виносяться назовні через подію onFiltersChange, а компонент Residents.vue уже працює з готовим об’єктом для Strapi. Цей підхід дозволяє тримати бізнес-логіку фільтрації в одному місці й спрощує її подальше розширення.

Також розглянемо реалізацію сторінки apply-to-join, де відбувається взаємодія одразу з двома різними backend-сервісами. Ця сторінка є одним з найбільш насичених прикладів інтеграції архітектурних рішень, адже вона поєднує в собі:

- Nuxt-сторінку з власним лейаутом (layout: ‘empty’),
- Комплекс UI-компонентів (Modal.vue, Input.vue, Checkbox.vue, Radio.vue, Button.vue, Popup.vue),
- Стан з Pinia-стора (notificationsStore, viewPortStore),
- Валідацію через yup та vee-validate,
- Два різних backend, а саме Strapi (збереження даних заявки) та Sendpulse (email-повідомлення).

Схема валідації формується динамічно в залежності від вибраного типу співпраці, що дозволяє суворо контролювати обов’язкові поля та одночасно не перевантажувати користувача зайвими полями для введення.

Після успішної валідації дані очищуються від порожніх значень, додатково перевіряються на відповідність типів та відправляються у Strapi через useJoinUsFormApi().fillJoinUsForm(), а також у Sendpulse для відправки листів користувачу та адміністратору. У випадку помилки використовується notificationsStore і компонент Notification.vue, який показує стандартизоване повідомлення. Повний код сторінки можна побачити в додатку Е.

Окремо в кваліфікаційній роботі було розглянуто архітектуру модулів Events та банерів, адже на їхньому прикладі ми можемо побачити як в проєкті поєднуються дані з API, кешування, скелетони та складні UI-патерни.

Секція Events використовує useEentsApi() та useAsyncData() для отримання майбутніх заходів з CMS. Фільтр по даті будується окремим хелпером buildDateFilter(), а в інтерфейсі реалізовано три стани:

- Скелетони під час завантаження,
- Перелік карток за їх наявності,
- Заглушка EmptyListPlaceholder при відсутності карток подій.

Банерний блок реалізований на базі Swiper з автопрокруткою, фракційною пагінацією та прогрес-баром. Додатковий прогрес-бар синхронізується з подією autoplayTimeLeft, а кнопка «Next» використовує прямий доступ до інстансу Swiper. Завдяки розділенню на окремі компоненти слайдер можна розширювати або змінювати без ризику породити «монолітний» компонент на декілька сотень рядків.

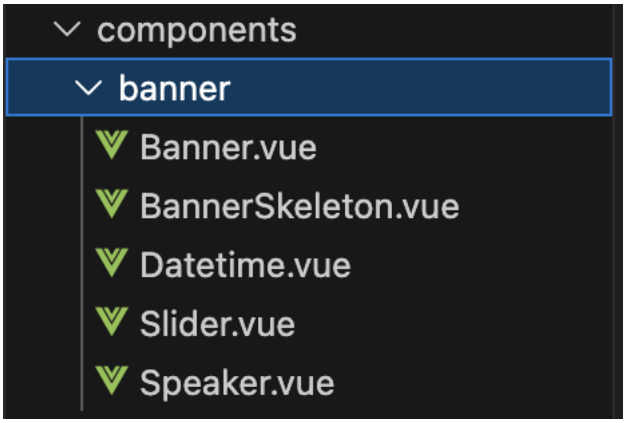

Рисунок 3.6 – Структура банерного блоку

У висновку можемо побачити, що фактична реалізація архітектури застосунку, виконана за допомогою Nuxt 3 показує, що ефективність фреймворку проявляється не лише у швидкому рендерингу, а насамперед у можливості побудувати чисту, керовану структуру:

1) Доступ до Strapi винесено в окремий прокші-шар server/api/strapi з кешуванням і централізованою автентифікацією,

2) API-модулі повертають уже опрацьовані дані, а не сирі HTTP-відповіді, що спрощує компоненти,

3) useAsyncData() з прапором lazy: true та getCechedData() використовуються лише там, де це справді дає виграш за рахунок кешу і повторного використання даних,

4) конфігураційний підхід до фільтрів резидентів зменшує обсяг коду та спрощує додавання нових критеріїв відбору,

5) форма apply-to-join демонструє, як можна поєднати валідацію, два бекенди та централізовану систему сповіщень у межах одного екосистемного рішення,

6) складні блоки на кшталт банерного слайдера реалізовано як набір дрібніших компонентів із чітким розподілом відповідальності.

У підсумку розроблена архітектура веб-застосунку підтверджує тезу та основну наукову гіпотезу кваліфікаційної роботи: грамотне використання сучасних веб-фреймворків (Vue 3 + Nuxt 3) дає змогу побудувати масштабований, керований та ефективний у підтримці веб-застосунок, не ускладнюючи кодову базу навіть при наявності складних інтеграцій та бізнес-логіки.

3.3 Перспективи покращень та подальшої оптимізації клієнтської частини проєкту

Поточна архітектура проєкту вже забезпечує суттєве зростання ефективності розробки та продуктивності застосунку: сторінки завантажуються швидше завдяки кешованим відповідям та миттєвим 304-відповідям, повторні переходи виконуються фактично без мережевих звернень, а скелетони роблять інтерфейс «живим» навіть до приходу реальних даних.

Каталог UI-компонентів дозволяє збирати нові секції значно швидше, а модульні форми легко розширюються без ризику зламати існуючу логіку. Однак навіть за наявності вже реалізованих оптимізацій існує низка напрямів, які можуть покращити продуктивність, масштабованість і підтримуваність проєкту надалі. Для прикладу:

1) Оптимізація роботи із зображеннями. Nuxt Image вже підключений, але поки не використовується. Перехід на нього зменшить вагу сторінок та прискорить відображення медіаконтенту.

2) Розширення composables. Надалі є сенс виділити універсальний механізм пагінації, універсальний composable для роботи з формами, загальний модуль для Strapi-filters. Це допоможе ще більше зменшити дублювання логіки в різних секціях.

3) Використання розширених можливостей Strapi. Створити більше динамічних компонентів в Strapi, аби менеджерський склад мав ще більше свободи при створенню наповнення для сайту.

Розроблена з використанням фреймворків архітектура проєкту за темою кваліфікаційної роботи вже принесла конкретні виграші: швидкі у завантажені сторінки, менше запитів, наявність миттєвих переходи, зручних форм та різке прискорення UI-розробки. Подальші покращення полягають не у переробленні системи, а у доведенні окремих модулів до рівня повної завершеності.

РОЗДІЛ 4.
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1.Обгрунтування можливих чинників травмонебезпечних ситуацій

У сучасних ІТ-компаніях характер виробничої діяльності переважно пов’язаний з інтелектуальною працею, використанням комп’ютерної техніки та тривалим перебуванням працівників у приміщеннях офісного типу. Незважаючи на відсутність важкого фізичного навантаження, умови праці в ІТ-сфері містять низку потенційно травмонебезпечних чинників, які можуть призводити до нещасних випадків, погіршення стану здоров’я або зниження працездатності персоналу.

До основних фізичних чинників травмонебезпечних ситуацій в ІТ-компаніях належать електротравми, пов’язані з експлуатацією комп’ютерного обладнання, мережевих пристроїв та систем безперебійного живлення. Пошкодження електропроводки, використання несправних розеток або перевантаження електромережі можуть створювати ризик ураження електричним струмом або виникнення пожежонебезпечних ситуацій.

Значну небезпеку становлять також ергономічні чинники, зокрема невідповідність робочого місця вимогам ергономіки. Неправильне розташування монітора, клавіатури, крісла або робочого столу може спричиняти порушення постави, перенапруження м’язів, захворювання опорно-рухового апарату та зорової системи. Тривале статичне навантаження без належних перерв підвищує ризик виникнення професійних захворювань і функціональних розладів.

До психофізіологічних чинників травмонебезпечних ситуацій належать високий рівень розумового навантаження, стрес, дедлайни та монотонність роботи. Хронічна втома та емоційне перенапруження можуть призводити до зниження концентрації уваги, що підвищує ймовірність помилок, нещасних

випадків та травм, зокрема при роботі з технічними засобами або в умовах обмеженого часу.

Окрему групу становлять організаційні чинники, пов’язані з неналежною організацією робочого процесу. Відсутність інструктажів з охорони праці, недотримання правил експлуатації обладнання, захаращені робочі проходи, погане освітлення або порушення норм мікроклімату можуть створювати додаткові ризики для безпеки працівників.

Таким чином, навіть за умов відносно безпечної офісної діяльності, IT-компанії мають потенційні чинники виникнення травмонебезпечних ситуацій. Їх своєчасне виявлення, аналіз та мінімізація шляхом впровадження організаційних, технічних і профілактичних заходів є важливою складовою системи охорони праці та забезпечення безпечних умов праці в IT-сфері.

4.2. Умови та обставини виникнення небезпечних ситуацій та їх наслідки

Умови праці в IT-компаніях здебільшого характеризуються хорошим станом робочого місця працівників в офісних приміщеннях та високим рівнем використання комп’ютерної техніки. За певних обставин можуть виникати небезпечні ситуації, що становлять загрозу для здоров’я та безпеки працівників. Такі ситуації формуються під впливом поєднання технічних, організаційних, ергономічних і психофізіологічних чинників.

Однією з основних умов виникнення небезпечних ситуацій є порушення правил експлуатації електротехнічного обладнання. Використання несправних електроприладів, перевантаження мережі, відсутність заземлення або пошкодження кабелів можуть призвести до ураження електричним струмом, коротких замикань чи виникнення пожеж. Особливо небезпечними є такі ситуації у приміщеннях із високою щільністю комп’ютерної техніки.

До небезпечних обставин також належить невідповідність робочих місць ергономічним вимогам. Тривале перебування працівників у вимушеній

або статичній позі, недостатнє або надмірне освітлення, неправильне розташування монітора та органів введення інформації створюють передумови для розвитку професійних захворювань, зниження працездатності та підвищення ризику травмування внаслідок перевтоми або втрати концентрації.

Вагомий вплив на виникнення небезпечних ситуацій мають психофізіологічні умови праці, зокрема високий рівень розумового навантаження, дефіцит часу на виконання завдань, тривалий стрес і порушення режиму праці та відпочинку. За таких обставин зростає ймовірність помилкових дій, неуважності та порушення правил безпеки, що може призводити як до технічних збоїв, так і до травмування працівників.

Організаційні недоліки, зокрема відсутність або формальний характер інструктажів з охорони праці, неналежний контроль за дотриманням норм безпеки, захаращені евакуаційні шляхи чи невідповідні умови мікроклімату, також створюють сприятливі умови для виникнення небезпечних ситуацій. У разі надзвичайних обставин це може ускладнити своєчасну евакуацію персоналу та збільшити тяжкість наслідків.

Наслідками небезпечних ситуацій в IT-компаніях можуть бути травми різного ступеня тяжкості, професійні захворювання опорно-рухового апарату та зорової системи, психоемоційне виснаження, а також матеріальні збитки, пов’язані з пошкодженням обладнання або перериванням робочих процесів. Отже, своєчасне виявлення умов і обставин виникнення небезпечних ситуацій та аналіз їх можливих наслідків є необхідною передумовою для розробки ефективних заходів з охорони праці в IT-компаніях, що дозволить створити безпечне та комфортне місце роботи як для працівників, так і відвідувачів офісу компанії.

4.3. Безпека в надзвичайних ситуаціях

В умовах військового стану забезпечення безпеки працівників набуває особливої актуальності та стає невід'ємною складовою системи охорони праці в організаціях, зокрема в ІТ-компаніях. Надзвичайні ситуації воєнного характеру, такі як повітряні тривоги, ракетні обстріли та загроза застосування вибухових засобів, створюють додаткові ризики для життя і здоров'я працівників та вимагають впровадження чітко регламентованих заходів безпеки.

Одним із ключових елементів забезпечення безпеки в надзвичайних ситуаціях є своєчасне інформування персоналу про загрозу. Використання офіційних систем оповіщення, внутрішніх корпоративних каналів зв'язку та відповідальних осіб дозволяє оперативно довести інформацію про початок повітряної тривоги та необхідність негайного припинення робочого процесу. Працівники повинні бути проінструктовані щодо алгоритму дій у разі отримання сигналу тривоги.

Важливе значення має наявність та належний стан захисних укриттів. ІТ-компанії зобов'язані забезпечити доступ працівників до найближчих сертифікованих укриттів або захисних споруд цивільного захисту, а також інформувати персонал про їх розташування та шляхи евакуації. Укриття повинні бути обладнані засобами першої медичної допомоги, аварійним освітленням, вентиляцією та мати достатній запас питної води.

Окрему увагу слід приділяти організації евакуації. Евакуаційні маршрути мають бути чітко позначені, вільні від перешкод і доступні для всіх працівників, зокрема осіб з обмеженими можливостями. Проведення регулярних інструктажів і навчальних тренувань сприяє зниженню паніки та забезпечує злагоджені дії персоналу в екстрених умовах.

З урахуванням специфіки ІТ-галузі важливим є також застосування гнучких форм організації праці, зокрема дистанційної роботи або змішаного формату, що дозволяє мінімізувати перебування працівників у потенційно

небезпечних зонах. У разі неможливості дистанційної роботи роботодавець має забезпечити безпечні умови перебування працівників у приміщеннях компанії.

Таким чином, безпека в надзвичайних ситуаціях в умовах військового стану передбачає комплексний підхід, який поєднує організаційні, технічні та інформаційні заходи. Реалізація цих заходів спрямована на збереження життя і здоров’я працівників, зниження ризиків травмування та забезпечення безперервності діяльності ІТ-компаній у складних умовах воєнного часу.

РОЗДІЛ 5.
ВИЗНАЧЕННЯ ЕФЕКТИВНОСТІ

Ефективність використання веб-фреймворків визначається тим, наскільки вони спрощують процес розробки, скорочують витрати часу, підвищують стабільність роботи застосунку та забезпечують можливість швидкого масштабування. Для оцінювання ефективності було проаналізовано роботу створеного веб-застосунку, який реалізований на основі фреймворку Vue та метафреймворку Nuxt. Аналіз охоплює технічні, організаційні та експлуатаційні аспекти, що дозволяють комплексно оцінити доцільність використання сучасних фреймворків у веб-розробці.

Одним із ключових критеріїв ефективності є прискорення завантаження інтерфейсу та підвищення якості користувацького досвіду. У розробленому застосунку для критичних секцій використовується частковий SSR-рендеринг та механізми кешування даних, що є прямим результатом застосування Nuxt. Завдяки цьому сторінки формуються значно швидше, ніж у традиційних рішеннях без фреймворків: структура інтерфейсу відображається миттєво, а дані з API довантажуються асинхронно. Компоненти-скелетони забезпечують візуальну завершеність сторінок незалежно від часу відповіді сервера. Це підвищує сприйняття швидкодії системи та робить роботу застосунку більш плавною.

Другим важливим аспектом є оптимізація кількості мережевих запитів. Фреймворк Nuxt забезпечує внутрішні механізми кешування результатів API-викликів. Це дає можливість зменшити кількість повторних запитів до сервера та забезпечити миттєве завантаження вже відвіданих сторінок. У розробленому застосунку після початкового отримання даних більшість повторних переходів не потребує додаткових мережевих звернень. Це дозволяє економити серверні ресурси, знижувати затримки та усувати зайве навантаження на інфраструктуру. Ефективність цього підходу особливо

відчутна у порівнянні з безфреймворковими сайтами, де кожне оновлення сторінки примусово генерує повний цикл запитів.

Значний внесок фреймворків полягає і в підвищенні ефективності процесу розробки. Створена бібліотека UI-компонентів, побудована на Vue, дає можливість багаторазово використовувати однакові елементи інтерфейсу в різних секціях сайту, не дублюючи код. Це стосується як базових елементів на кшталт Input, Checkbox, Button, так і складніших структур — фільтрів, навігаційних компонентів, банерних слайдерів. У традиційній розробці без фреймворків підтримка великої кількості елементів вимагала б створення окремих реалізацій для кожної сторінки, що ускладнює внесення змін та збільшує кількість потенційних помилок. Натомість компонентний підхід забезпечує масштабованість, повторюваність та стабільність інтерфейсу

Окрім цього, використання метафреймворку дає змогу структуровано організувати архітектуру проєкту. Nuxt автоматично формує маршрути, забезпечує єдину логіку для роботи з API, додає модульність і впорядкованість. Це зменшує складність підтримки проєкту, спрощує онбординг нових розробників та дозволяє розширювати функціонал без ризику порушення існуючої логіки. Зокрема, додавання нових секцій або форм у розробленому застосунку потребує мінімальних зусиль, оскільки значна частина логіки реалізована у вигляді composables та повторно використовуваних модулів.

Технічна ефективність також проявляється у показниках продуктивності застосунку. Завдяки кешуванню, оптимізації запитів та використанню реактивної моделі Vue, система демонструє стабільну роботу навіть при збільшенні кількості елементів на сторінці. Фільтри, списки та форми працюють без затримок, а взаємодія користувачів із застосунком залишається плавною.

Важливим є й аспект економічної ефективності. Скорочення часу розробки, повторне використання компонентів, автоматизована маршрутизація та стандартизована архітектура дають змогу зменшити витрати

на створення та підтримку застосунку. У порівнянні з ручною розробкою або сайтами, створеними у конструкторах з обмеженою гнучкістю, застосування фреймворків забезпечує баланс між швидкістю реалізації та високою якістю результату (додаток Є).

Проведений аналіз показує, що використання сучасних веб-фреймворків суттєво підвищує якість застосунку, прискорює його розробку та забезпечує технологічну стабільність. Практичні результати, отримані під час створення клієнтської частини веб-застосунку, підтверджують, що Vue та Nuxt є ефективними інструментами, які дозволяють будувати масштабовані, продуктивні та зручні у підтримці системи. Це доводить доцільність застосування фреймворків у сучасній веб-розробці та їхнє значення для створення високоякісних користувацьких інтерфейсів.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи було досліджено теоретичні та практичні аспекти використання сучасних клієнтських фреймворків при створенні веб-застосунків, а також проаналізовано їх вплив на ефективність розробки, продуктивність та масштабованість клієнтської частини сайту. Основну увагу було приділено компонентному підходу, реактивній моделі програмування та сучасним архітектурним рішенням, що застосовуються у фронтенд-розробці, зокрема клієнтської частини проекту.

У межах роботи обґрунтовано доцільність використання фреймворку Vue як основи клієнтської частини веб-застосунку завдяки його декларативній моделі, вбудованій реактивності та низькому порогу входу. Застосування компонентної архітектури дозволило забезпечити модульність, повторне використання інтерфейсних елементів і зручність супроводу коду, що є критично важливим для масштабованих веб-систем.

Використання метафреймворку Nuxt 3 дало змогу реалізувати сучасні підходи до рендерингу сторінок, зокрема гібридного підходу для режиму відображення. Це сприяло підвищенню продуктивності клієнтської частини, покращенню часу першого завантаження сторінок і забезпеченню кращої індексації веб-ресурсу пошуковими системами. Крім того, вбудовані механізми маршрутизації та керування станом значно спростили процес розробки та зменшили кількість конфігураційних налаштувань.

Для організації роботи з даними було використано Strapi як headless CMS, що дозволило відокремити клієнтську частину від серверної логіки та реалізувати гнучку взаємодію через API. Такий підхід забезпечив незалежність розвитку фронтенду та бекенду, спростив керування контентом і підвищив адаптивність веб-застосунку до змін бізнес-вимог.

Практична реалізація клієнтської частини веб-застосунку із застосуванням стеку Vue + Nuxt 3 + Strapi підтвердила ефективність обраних

технологій для створення сучасних, продуктивних і зручних у супроводі веб-рішень. Отримані результати засвідчили, що використання фреймворків і метафреймворків суттєво скорочує час розробки, підвищує якість коду та покращує користувацький досвід.

Реалізація поставлених завдань та розроблена архітектура веб-застосунку підтвердила на практиці, що грамотне використання сучасних веб-фреймворків (Vue 3 + Nuxt 3) дає змогу побудувати масштабований, керований та ефективний у підтримці веб-застосунок, не ускладнюючи кодову базу навіть при наявності складних інтеграцій та бізнес-логіки.

Результати, отримані під час виконання кваліфікаційної роботи, можуть бути використані в подальших наукових дослідженнях, а також у практичній діяльності при розробці клієнтської частини веб-застосунків різного рівня складності. Перспективним напрямом подальших досліджень є порівняльний аналіз ефективності інших клієнтських фреймворків і метафреймворків, а також дослідження впливу різних стратегій рендерингу на продуктивність і масштабованість веб-застосунків.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Філімончук Т.В., Колтун Ю.М., Климова І.М., Холобок В.І. Модель підходу розробки мобільних застосунків. *Вчені записки ТНУ ім.Вернадського. Серія: Технічні науки*. Том 36 (75) № 3 2025. С. 429-436.

2. Босько В.В., Константинова Л.В., Марченко К.М., Улічев О.С. Web-програмування. Частина 1 (frontend): навч. посіб. Кропивницький: ЦНТУ, 2022. 208 с.

3. O'Grady B. Client-Side vs Server-Side. URL: https://codeinstitute.net/global/blog/client-side-vs-server-side/ (дата звернення 21.08.2025).

4. Iskandar T., Lubis M., Kusumasari T., Lubis A. Comparison between client-side and server-side rendering in the web development. IOP Conf. Ser.: Mater. Sci. Eng., 2020. doi:10.1088/1757-899X/801/1/012136

5. Lokhande J., Mantri Sh., Hande Y. A Comparative Analysis of Client Side Rendering and Server Side Rendering. 2025. http://dx.doi.org/10.2139/ssrn.5832362

6. What’s the meaning of client side and server side? URL: https://www.debugbear.com/docs/client-vs-server-side (дата звернення 25.08.2025).

7. Introduction to client-side frameworks. URL: https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Frameworks_libraries/Introduction (дата звернення 22.08.2025).

8. Lazuardy M., Anggraini D. Modern Front End Web Architectures with React.Js and Next.Js. *International Research Journal of Advanced Engineering and Science.* URL: https://irjaes.com/wp-content/uploads/2022/02/IRJAES-V7N1P162Y22.pdf (дата звернення 12.07.2025).

9. Основи реактивного програмування та їх застосування у веб-розробці. URL: https://it-rating.ua/osnovi-reaktivnogo-programuvannya-ta-ih-zastosuvannya-v-veb-rozrobtsi (дата звернення 14.08.2025).

10. Holmes B. Understanding the JavaScript Meta-framework Ecosystem. URL: https://prismic.io/blog/javascript-meta-frameworks-ecosystem (дата звернення 07.09.2025).

11. Oliveira J., Alarcão S.M., Chambel T., Forseca M. MetaFERA: a meta-framework for creating emotion recognition frameworks for physiological signals.*Multimed Tools Appl.* 83, 2024. 9785–9815. https://doi.org/10.1007/s11042-023-15249-5

12. JavaScript Meta Framework. URL: https://sailssoftware.com/javascript-meta-framework/ (дата звернення 16.08.2025).

13. Build a Flutter App using MVVM. URL: https://medium.com/@DevMonarch/implementing-model-view-viewmodel-mvvm-in-a-flutter-app-a-step-by-step-guide-92b05e6e8192 (дата звернення 10.09.2025).

14. Mokxtari M. The Evolution of JavaScript Frameworks: From jQuery to Svelte, with React and Vue Along the Way. URL: https://medium.com/@mohamed.mokhtari/the-evolution-of-javascript-frameworks-from-jquery-to-svelte-with-react-and-vue-along-the-way-a5393fc0f628 (дата звернення 16.08.2025).

15. Bolic M. SSR with Vue.js (Part 2) — Setting Up State Management with SSR. URL: https://medium.com/@mladen.bolic/ssr-with-vue-js-part-2-set-up-state-management-with-ssr-5427d65d3cfb (дата звернення 12.09.2025).

16. React vs Angular: дві сторони JS. URL: https://itvdn.com/ua/blog/article/reactvsangular (дата звернення 14.07.2025).

17. Hamsa M. Advantages of Developing Modern Web apps with React.js. URL: https://hamzamahmood.medium.com/advantages-of-developing-modern-web-apps-with-react-js-8504c571db71 (дата звернення 17.07.2025).

18. Andrzejewski J. Understanding the Vue 3 Composition API. URL: https://dev.to/jacobandrewsky/understanding-the-vue-3-composition-api-fa2 (дата звернення 17.09.2025).

19. Andrzejewski P. From Vue to Nuxt: Server-side rendering in a nutshell. URL: https://alokai.com/blog/vue-nuxt-server-side-rendering (дата звернення 21.09.2025).

20. Andrzejewski J. How To Optimize Performance In Nuxt Apps. URL: https://www.debugbear.com/blog/optimize-nuxt-performance (дата звернення 15.10.2025).

21. Dixit P. React vs Angular vs Vue – select the best front end technology for your web app. URL: https://www.tristatetechnology.com/blog/react-vs-angular-vs-vue-comparison (дата звернення 17.07.2025).

22. Тимочко В.О., Городецький І.М., Березовецький А.П., Ковальчук Ю.О., Мазур І.Б., Стефанишин В.Ю. Безпека життєдіяльності та охорона праці. Практикум: навч. посіб. Львів: Сполом. 2022. 336 с.

23. Nuxt: The Progressive Web Framework. URL: https://nuxt.com/ (дата звернення 12.07.2025).

24. Vue.js - The Progressive JavaScript Framework URL: https://vuejs.org/ (дата звернення 14.07.2025).

ДОДАТКИ

Додаток А

components/ui/Notification.vue

```
<template>
<div :class="notificationClasses">
<IconsCross class="ui-notification__close-icon" @click="onCloseNotification"/>
<p v-if="notificationData.title" class="ui-notification__title">
{{ notificationData.title }}
</p>
<p v-if="notificationData.message" class="ui-notification__message">
{{ notificationData.message }}
</p>
</div>
</template>
<script setup lang="ts">
import type { INotification } from '~/types';
interface Props {
notificationData: INotification;
}
const props = defineProps<Props>();
const emit = defineEmits(['onCloseNotification']);
const notificationClasses = computed(() => {
const basicClass = 'ui-notification';
const classes = [
basicClass,
basicClass + `--${props.notificationData.type}`,
];
return classes;
});
onMounted(() => {
setTimeout(() => {
onCloseNotification();
}, 5000);
});
function onCloseNotification() {
emit('onCloseNotification', props.notificationData.id);
}
</script>
<style lang="scss">
.ui-notification {
position: relative;
width: 340px;
display: flex;
flex-direction: column;
gap: 8px;
padding: 32px;
&--info {
background-color: $aqua-25;
}
&--success {
```

```
background-color: $green-25;
}
&--warning {
background-color: $yellow-25;
}
&--error {
background-color: $red-25;
}
@include phone {
width: 100%;
padding: 28px;
}
&__close-icon {
position: absolute;
inset-block-start: 16px;
inset-inline-end: 16px;
cursor: pointer;
}
&__title {
text-transform: uppercase;
font-size: 16px;
line-height: 1;
font-weight: 700;
@include phone {
font-size: 14px;
}
}
```

Додаток Б

components/ui/Chip.vue

```
<template>
<div class="ui-chip">
<span class="ui-chip__label">
{{ option.label }}
</span>
<IconsCloseFilled
class="ui-chip__close"
@click="onChipClose"
/>
</div>
</template>
<script setup lang="ts">
import type { IChipOption } from '~/types';
interface Props {
option: IChipOption;
}
const props = defineProps<Props>();
const emit = defineEmits<{
(emitEvent: 'onClose', option: IChipOption): void,
}>();
function onChipClose() {
emit('onClose', props.option);
}
</script>
<style lang="scss">
.ui-chip {
display: flex;
align-items: center;
gap: 4px;
padding-block: 8px;
padding-inline-start: 16px;
padding-inline-end: 8px;
background-color: $salad;
border-radius: 60px;
transition: background-color .3s, color .3s;
@include hoverable {
&:has(.ui-chip__close:hover) {
color: $white;
background-color: $black;
}
&:has(.ui-chip__close:active) {
background-color: $gray-75;
}
}
@include phone {
padding-block: 6px;
padding-inline-start: 12px;
padding-inline-end: 6px;
}
```

```
&__label {
font-size: 15px;
line-height: 1.4;
@include phone {
font-size: 12px;
}
}
&__close {
cursor: pointer;
}
}
</style>
```

Додаток В

components/sections/residents/Residents.vue

```
<template>
<section class="residents-section">
<UiSectionHeader
class="residents-section__header container"
:smallPart="t('residentsSection.meetOurXrResidents.part1')"
:mainPart="t('residentsSection.meetOurXrResidents.part2')"
/>
<SectionsResidentsFilters
class="residents-section__filters container"
:areFiltersDisabled="!areResidentsLoaded || isMoreLoadingActive"
@onFiltersChange="updateCurrentFilters"
/>
<UiEmptyListPlaceholder
v-if="areResidentsLoaded && residents && residents.length === 0"
class="container"
:title="t('shared.emptyHere')"
:message="t('residentsSection.noResidentsMatchYourFilters')"
/>
<div
v-else
class="residents-section__residents container"
:class="{
'residents-section__residents--with-button': currentPagination.page !==
currentPagination.pageCount,
}"
>
<template v-if="!areResidentsLoaded && showSkeleton">
<SectionsResidentsSkeletonCard
v-for="n in pageSize"
:key="n"
class="residents-section__resident"
/>
</template>
<template v-else-if="areResidentsLoaded">
<SectionsResidentsCard
v-for="resident in residents"
:key="resident.id"
:residentData="resident"
class="residents-section__resident"
/>
</template>
</div>
<UiLoader
v-if="!areResidentsLoaded && !showSkeleton"
class="residents-section__loader"
size="lg"
```

```
/>
<div
v-if="
areResidentsLoaded
&& currentPagination.pageCount
&& currentPagination.pageCount > 0
&& currentPagination.page !== currentPagination.pageCount
"
class="residents-section__load-more-container container"
>
<UiButton
class="residents-section__load-more-button"
:isLoading="isMoreLoadingActive"
@click="loadMore"
>
{{ t('shared.loadMore') }}
<template #icon>
<IconsThreeDash />
</template>
</UiButton>
</div>
</section>
</template>
<script setup lang="ts">
import type { IResident, IResidentPopulate, IStrapiFilters, IStrapiPagination, IStrapiSort } from
'~/types';
import { useResidentsApi } from '~/api/strapi/residents.api';
import { BREAKPOINTS } from '~/constants';
const { t } = useI18n();
const viewPortStore = useViewPortStore();
const { getResidents } = useResidentsApi();
const residents = ref<IResident[]>([]);
const showSkeleton = ref(true);
const areResidentsLoaded = ref(false);
const isMoreLoadingActive = ref(false);
const currentFilters = ref<IStrapiFilters>({});
const currentPagination = ref<Partial<IStrapiPagination>>({});
const currentPage = ref(1);
const mobilePageSize = 4;
const desktopPageSize = 16;
const pageSize = ref(viewPortStore.width <= BREAKPOINTS.SMALL_TABLET ?
mobilePageSize : desktopPageSize);
async function loadResidents(
filters: IStrapiFilters,
pagination: Partial<IStrapiPagination>,
isLoadMore = false,
) {
if (isLoadMore) {
isMoreLoadingActive.value = true;
} else {
areResidentsLoaded.value = false;
}
```

```
const residentsPopulate: IResidentPopulate = {
image: true,
member_type: true,
company_type: true,
product_type: true,
company_size: true,
services: true,
industries: true,
}
const residentsSort: IStrapiSort<IResident> = { createdAt: 'desc' };
const res = (await getResidents(
residentsPopulate,
filters,
pagination,
residentsSort,
));
if (isLoadMore) {
residents.value = [...residents.value, ...res.data];
isMoreLoadingActive.value = false;
} else {
residents.value = res.data;
areResidentsLoaded.value = true;
}
if (showSkeleton.value) {
showSkeleton.value = false;
}
currentPagination.value = res.meta.pagination;
}
function updateCurrentFilters(filters: IStrapiFilters) {
currentFilters.value = filters;
}
function loadMore() {
currentPage.value = currentPage.value + 1;
}
watch(currentFilters, async (newFilters) => {
currentPage.value = 1;
await loadResidents(newFilters, { page: currentPage.value, pageSize: pageSize.value });
});
watch(currentPage, async (newPage) => {
if (newPage !== 1) {
await loadResidents(currentFilters.value, { page: newPage, pageSize: pageSize.value }, true);
}
});
watch(() => viewPortStore.width, async (newVal) => {
if (newVal <= BREAKPOINTS.SMALL_TABLET && pageSize.value === desktopPageSize) {
pageSize.value = mobilePageSize;
currentPage.value = 1;
await loadResidents(currentFilters.value, { page: currentPage.value, pageSize: pageSize.value
});
} else if (newVal > BREAKPOINTS.SMALL_TABLET && pageSize.value !==
desktopPageSize) {
pageSize.value = desktopPageSize;
```

```
currentPage.value = 1;
await loadResidents(currentFilters.value, { page: currentPage.value, pageSize: pageSize.value
});
}
});
</script>
<style lang="scss">
.residents-section {
overflow-x: clip;
&__header {
margin-bottom: 48px;
@include tablet {
margin-bottom: 40px;
}
}
&__loader {
min-height: 400px;
}
&__filters {
display: flex;
flex-direction: column;
gap: 4px;
margin-bottom: 32px;
}
&__residents {
display: grid;
// TODO: make min width 320px and remove double border
grid-template-columns: repeat(auto-fill, minmax(320px, 1fr));
@include phone {
padding-inline: 0;
}
&--with-button {
margin-block-end: 48px;
@include phone {
margin-block-end: 32px;
}
}
}
&__resident {
min-height: 450px;
// TODO: is it the best way to remove double border?
margin-inline-end: -1px;
margin-block-end: -1px;
@include phone {
min-height: 400px;
border: none;
margin-inline-end: 0;
margin-block-end: 0;
&:first-child {
&::before {
content: '';
position: absolute;
```

```
inset-block-start: 0;
inset-inline-start: 20px;
inset-inline-end: 20px;
height: 1px;
background-color: $black;
}
}
&::after {
content: '';
position: absolute;
inset-block-end: 0;
inset-inline-start: 20px;
inset-inline-end: 20px;
height: 1px;
background-color: $black;
}
&:not(:first-child) {
margin-block-start: -1px;
}
}
}
&__load-more-container {
display: flex;
justify-content: center;
}
&__load-more-button {
width: 312px;
@include small-tablet {
width: 100%;
}
}
}
</style>
```

Додаток Г

components/sections/residents/Filters.vue

```
<template>
<div
class="resident-card"
:class="{ 'resident-card--active': shouldShowOverlay }"
@mouseleave="handleOverlayVisibility(false)"
>
<Transition name="resident-card-overlay">
<div
v-if="shouldShowOverlay"
class="resident-card__content resident-card__content--overlay"
>
<div class="resident-card__header resident-card__header--overlay">
<p class="resident-card__title resident-card__title--overlay">
{{ residentData.name }}
</p>
<button
v-if="!isHoverable"
class="resident-card__toggle-button resident-card__toggle-button--overlay"
@click="handleMobileOverlayVisibility(false)"
>
<IconsPlus />
</button>
</div>
<div
v-if="residentData.member_type?.slug === 'service' && isValidArray(residentData.services)"
class="resident-card__tags-container resident-card__tags-container--overlay"
>
<p class="resident-card__tags-title resident-card__tags-title--overlay">
{{ t('shared.services') }}
</p>
<div class="resident-card__tags">
<UiTag
v-for="service in residentData.services?.slice(0, 5)"
:key="service.id"
:label="service.name"
:design="tagsDesign"
outlined
/>
</div>
</div>
<div v-if="isValidArray(residentData.industries)" class="resident-card__tags-container resident-
card__tags-container--overlay">
<p class="resident-card__tags-title resident-card__tags-title--overlay">
{{ t('shared.industries', 2) }}
</p>
<div class="resident-card__tags">
<UiTag
v-for="industry in residentData.industries?.slice(0, 5)"
:key="industry.id"
```

```
:label="industry.name"
:design="tagsDesign"
outlined
/>
</div>
</div>
</div>
</Transition>
<div
class="resident-card__content resident-card__content--able-hover"
@mouseenter="handleOverlayVisibility(true)"
>
<div class="resident-card__header">
<img
class="resident-card__logo"
:src="residentData.image?.url"
:alt="getResidentLogoAltText()"
>
<button
v-if="!isHoverable"
class="resident-card__toggle-button"
@click="handleMobileOverlayVisibility(true)"
>
<IconsPlus />
</button>
</div>
<p class="resident-card__title">
{{ residentData.name }}
</p>
<p class="resident-card__info">
{{ residentData.info }}
</p>
</div>
<div class="resident-card__content resident-card__content--disable-hover">
<div class="resident-card__tags-container">
<div class="resident-card__tags">
<UiTag
v-if="residentData.member_type"
:label="residentData.member_type.name"
:design="tagsDesign"
/>
<UiTag
v-if="residentData.company_type"
:label="residentData.company_type.name"
:design="tagsDesign"
/>
<UiTag
v-if="residentData.member_type?.slug === 'product' && residentData.product_type"
:label="residentData.product_type.name"
:design="tagsDesign"
/>
<UiTag
```

```
v-if="residentData.member_type?.slug === 'service' && residentData.company_size"
:label="residentData.company_size.name"
:design="tagsDesign"
/>
</div>
</div>
<a
:href="residentData.link"
alt="arrow"
class="resident-card__visit"
target="_blank"
>
<UiButton :size="EButtonSize.SM">
{{ t('shared.visit') }}
<template #icon>
<IconsArrowRight class="resident-card__visit-icon" />
</template>
</UiButton>
</a>
</div>
</div>
</template>
<script setup lang="ts">
import { EButtonSize, type IResident } from '~/types';
import { isValidArray } from '~/helpers';
interface Props {
residentData: IResident;
}
const props = defineProps<Props>();
const { t } = useI18n();
const { isHoverable } = usePointerMode();
const isOverlayContentVisible = ref(false);
function handleOverlayVisibility(visibility: boolean) {
isOverlayContentVisible.value = visibility;
}
const isMobileOverlayContentVisible = ref(false);
function handleMobileOverlayVisibility(visibility: boolean) {
isMobileOverlayContentVisible.value = visibility;
}
const shouldShowOverlay = computed(() => {
return isHoverable.value
? isOverlayContentVisible.value
: isMobileOverlayContentVisible.value;
});
function getResidentLogoAltText() {
let alt = props.residentData.name;
if (props.residentData.name.length > 22) {
alt = props.residentData.name.slice(0, 22).trim() + '...';
}
return alt + ' (logo)';
}
```

```
const tagsDesign = props.residentData.member_type?.slug === 'product' ? 'secondary' :
'primary';
</script>
<style lang="scss">
.resident-card {
position: relative;
display: flex;
flex-direction: column;
border: 1px solid black;
transition: clip-path .3s ease, -webkit-clip-path .3s ease;
@include cut-angles;
&--active {
@include cut-angles(0px);
& > *:not(.resident-card__content--overlay) {
pointer-events: none;
}
}
@include phone {
@include cut-angles(0px);
}
&__toggle-button {
background-color: transparent;
border: none;
margin-block-end: auto;
&--overlay {
color: $aqua;
transform: rotateZ(45deg);
}
}
&__content {
display: flex;
flex-direction: column;
padding: 24px;
@include phone {
padding: 20px;
}
&--overlay {
position: absolute;
inset: 0;
gap: 24px;
background-color: $black;
@include phone {
gap: 20px;
}
}
&--able-hover {
flex-grow: 1;
padding-bottom: 8px;
}
&--disable-hover {
padding-top: 8px;
}
```

```
}
&__header {
display: flex;
justify-content: space-between;
align-items: center;
height: 72px;
margin-block-end: 20px;
&--overlay {
height: auto;
margin-block-end: 0;
}
}
&__logo {
max-height: 100%;
}
&__title {
text-transform: uppercase;
font-size: 20px;
line-height: 1.4;
font-weight: 700;
margin-block-end: 12px;
&--overlay {
color: $white;
line-height: 1.2;
margin-block-end: 0;
}
}
&__info {
color: $gray-75;
font-size: 15px;
line-height: 1.4;
}
&__tags-container {
margin-bottom: 24px;
&--overlay {
margin-bottom: 0;
}
}
&__tags-title {
font-size: 15px;
line-height: 1.4;
font-weight: 700;
margin-bottom: 8px;
&--overlay {
color: $white;
}
}
&__tags {
display: flex;
gap: 4px;
flex-wrap: wrap;
}
```

```
&__visit {
width: min-content;
margin-top: auto;
}
&__visit-icon {
transform: rotateZ(315deg);
}
}
.resident-card-overlay-enter-active,
.resident-card-overlay-leave-active {
transition: opacity 0.3s ease;
}
.resident-card-overlay-enter-from,
.resident-card-overlay-leave-to {
opacity: 0;
}
.resident-card-overlay-enter-to,
.resident-card-overlay-leave-from {
opacity: 1;
}
</style>
```

Додаток Ґ

components/sections/residents/Card.vue

```
<template>
<div
class="resident-card"
:class="{ 'resident-card--active': shouldShowOverlay }"
@mouseleave="handleOverlayVisibility(false)"
>
<Transition name="resident-card-overlay">
<div
v-if="shouldShowOverlay"
class="resident-card__content resident-card__content--overlay"
>
<div class="resident-card__header resident-card__header--overlay">
<p class="resident-card__title resident-card__title--overlay">
{{ residentData.name }}
</p>
<button
v-if="!isHoverable"
class="resident-card__toggle-button resident-card__toggle-button--overlay"
@click="handleMobileOverlayVisibility(false)"
>
<IconsPlus />
</button>
</div>
<div
v-if="residentData.member_type?.slug === 'service' && isValidArray(residentData.services)"
class="resident-card__tags-container resident-card__tags-container--overlay"
>
<p class="resident-card__tags-title resident-card__tags-title--overlay">
{{ t('shared.services') }}
</p>
<div class="resident-card__tags">
<UiTag
v-for="service in residentData.services?.slice(0, 5)"
:key="service.id"
:label="service.name"
:design="tagsDesign"
outlined
/>
</div>
</div>
<div v-if="isValidArray(residentData.industries)" class="resident-card__tags-container resident-
card__tags-container--overlay">
<p class="resident-card__tags-title resident-card__tags-title--overlay">
{{ t('shared.industries', 2) }}
</p>
<div class="resident-card__tags">
<UiTag
v-for="industry in residentData.industries?.slice(0, 5)"
:key="industry.id"
```

```
:label="industry.name"
:design="tagsDesign"
outlined
/>
</div>
</div>
</div>
</Transition>
<div
class="resident-card__content resident-card__content--able-hover"
@mouseenter="handleOverlayVisibility(true)"
>
<div class="resident-card__header">
<img
class="resident-card__logo"
:src="residentData.image?.url"
:alt="getResidentLogoAltText()"
>
<button
v-if="!isHoverable"
class="resident-card__toggle-button"
@click="handleMobileOverlayVisibility(true)"
>
<IconsPlus />
</button>
</div>
<p class="resident-card__title">
{{ residentData.name }}
</p>
<p class="resident-card__info">
{{ residentData.info }}
</p>
</div>
<div class="resident-card__content resident-card__content--disable-hover">
<div class="resident-card__tags-container">
<div class="resident-card__tags">
<UiTag
v-if="residentData.member_type"
:label="residentData.member_type.name"
:design="tagsDesign"
/>
<UiTag
v-if="residentData.company_type"
:label="residentData.company_type.name"
:design="tagsDesign"
/>
<UiTag
v-if="residentData.member_type?.slug === 'product' && residentData.product_type"
:label="residentData.product_type.name"
:design="tagsDesign"
/>
<UiTag
```

```
v-if="residentData.member_type?.slug === 'service' && residentData.company_size"
:label="residentData.company_size.name"
:design="tagsDesign"
/>
</div>
</div>
<a
:href="residentData.link"
alt="arrow"
class="resident-card__visit"
target="_blank"
>
<UiButton :size="EButtonSize.SM">
{{ t('shared.visit') }}
<template #icon>
<IconsArrowRight class="resident-card__visit-icon" />
</template>
</UiButton>
</a>
</div>
</div>
</template>
<script setup lang="ts">
import { EButtonSize, type IResident } from '~/types';
import { isValidArray } from '~/helpers';
interface Props {
residentData: IResident;
}
const props = defineProps<Props>();
const { t } = useI18n();
const { isHoverable } = usePointerMode();
const isOverlayContentVisible = ref(false);
function handleOverlayVisibility(visibility: boolean) {
isOverlayContentVisible.value = visibility;
}
const isMobileOverlayContentVisible = ref(false);
function handleMobileOverlayVisibility(visibility: boolean) {
isMobileOverlayContentVisible.value = visibility;
}
const shouldShowOverlay = computed(() => {
return isHoverable.value
? isOverlayContentVisible.value
: isMobileOverlayContentVisible.value;
});
function getResidentLogoAltText() {
let alt = props.residentData.name;
if (props.residentData.name.length > 22) {
alt = props.residentData.name.slice(0, 22).trim() + '...';
}
return alt + ' (logo)';
}
```

```
const tagsDesign = props.residentData.member_type?.slug === 'product' ? 'secondary' :
'primary';
</script>
<style lang="scss">
.resident-card {
position: relative;
display: flex;
flex-direction: column;
border: 1px solid black;
transition: clip-path .3s ease, -webkit-clip-path .3s ease;
@include cut-angles;
&--active {
@include cut-angles(0px);
& > *:not(.resident-card__content--overlay) {
pointer-events: none;
}
}
@include phone {
@include cut-angles(0px);
}
&__toggle-button {
background-color: transparent;
border: none;
margin-block-end: auto;
&--overlay {
color: $aqua;
transform: rotateZ(45deg);
}
}
&__content {
display: flex;
flex-direction: column;
padding: 24px;
@include phone {
padding: 20px;
}
&--overlay {
position: absolute;
inset: 0;
gap: 24px;
background-color: $black;
@include phone {
gap: 20px;
}
}
&--able-hover {
flex-grow: 1;
padding-bottom: 8px;
}
&--disable-hover {
padding-top: 8px;
}
```

```
}
&__header {
display: flex;
justify-content: space-between;
align-items: center;
height: 72px;
margin-block-end: 20px;
&--overlay {
height: auto;
margin-block-end: 0;
}
}
&__logo {
max-height: 100%;
}
&__title {
text-transform: uppercase;
font-size: 20px;
line-height: 1.4;
font-weight: 700;
margin-block-end: 12px;
&--overlay {
color: $white;
line-height: 1.2;
margin-block-end: 0;
}
}
&__info {
color: $gray-75;
font-size: 15px;
line-height: 1.4;
}
&__tags-container {
margin-bottom: 24px;
&--overlay {
margin-bottom: 0;
}
}
&__tags-title {
font-size: 15px;
line-height: 1.4;
font-weight: 700;
margin-bottom: 8px;
&--overlay {
color: $white;
}
}
&__tags {
display: flex;
gap: 4px;
flex-wrap: wrap;
}
```

```
&__visit {
width: min-content;
margin-top: auto;
}
&__visit-icon {
transform: rotateZ(315deg);
}
}
.resident-card-overlay-enter-active,
.resident-card-overlay-leave-active {
transition: opacity 0.3s ease;
}
.resident-card-overlay-enter-from,
.resident-card-overlay-leave-to {
opacity: 0;
}
.resident-card-overlay-enter-to,
.resident-card-overlay-leave-from {
opacity: 1;
}
</style>
```

Додаток Д

components/sections/residents/SkeletonCard.vue

```
<template>
<div class="resident-card-skeleton">
<UiSkeletonItem class="resident-card-skeleton__top" />
<div class="resident-card-skeleton__content">
<UiSkeletonItem class="resident-card-skeleton__content-top" />
<UiSkeletonItem class="resident-card-skeleton__content-middle" />
<div class="resident-card-skeleton__content-bottom-wrapper">
<UiSkeletonItem v-for="n in 3" :key="n" class="resident-card-skeleton__content-bottom" />
</div>
</div>
<UiSkeletonItem class="resident-card-skeleton__bottom" />
</div>
</template>
<script setup lang="ts">
</script>
<style lang="scss">
.resident-card-skeleton {
position: relative;
display: flex;
flex-direction: column;
padding: 24px;
border: 1px solid black;
@include cut-angles;
@include phone {
padding: 20px;
}
&__top {
height: 72px;
margin-block-end: 20px;
}
&__content {
flex-grow: 1;
}
&__content-top {
height: 15px;
margin-block-end: 12px;
}
&__content-middle {
height: 150px;
margin-block-end: 16px;
}
&__content-bottom-wrapper {
display: flex;
gap: 4px;
}
&__content-bottom {
height: 25px;
width: 70px;
```

```
display: flex;
gap: 4px;
}
&__bottom {
height: 40px;
width: 90px;
}
}
</style>
```

Додаток Е

pages/apply-to-join.vue

```
<template>
<div class="apply-to-join-page">
<UiModal
class="apply-to-join-page__modal"
:title="t('applyToJoin.applyToJoin')"
@onCloseModal="closeModal"
>
<form class="apply-to-join-page__form" @submit.prevent="onSubmit">
<div class="apply-to-join-page__radios">
<UiRadio
:label="t('shared.xrCompany')"
id="xrCompanyApplyToJoinRadio"
name="typeOfCooperation"
:value="EJoinUsTypeOfCooperationRadios.COMPANY"
v-model="typeOfCooperation"
:light="viewPortStore.width <= BREAKPOINTS.PHONE"
>
{{ t('shared.xrCompany') }}
</UiRadio>
<UiRadio
:label="t('shared.xrProfessional')"
id="xrProfessionalApplyToJoinRadio"
name="typeOfCooperation"
:value="EJoinUsTypeOfCooperationRadios.PROFESSIONAL"
v-model="typeOfCooperation"
:light="viewPortStore.width <= BREAKPOINTS.PHONE"
>
{{ t('shared.xrProfessional') }}
</UiRadio>
<UiRadio
:label="t('shared.partners', 1)"
id="xrPartnerApplyToJoinRadio"
```

```
name="typeOfCooperation"
:value="EJoinUsTypeOfCooperationRadios.PARTNER"
v-model="typeOfCooperation"
:light="viewPortStore.width <= BREAKPOINTS.PHONE"
>
{{ t('shared.partners', 1) }}
</UiRadio>
</div>
<div class="apply-to-join-page__inputs">
<UiInput
:required="true"
:placeholder="t('shared.yourFullName')"
:light="viewPortStore.width <= BREAKPOINTS.PHONE"
:invalid="!!fullNameError"
:errorMessage="fullNameError"
v-model.trim="fullName"
></UiInput>
<UiInput
v-if="typeOfCooperation === EJoinUsTypeOfCooperationRadios.COMPANY ||
typeOfCooperation === EJoinUsTypeOfCooperationRadios.PARTNER"
:required="true"
:placeholder="t('applyToJoin.companyName')"
:light="viewPortStore.width <= BREAKPOINTS.PHONE"
:invalid="!!companyNameError"
:errorMessage="companyNameError"
v-model.trim="companyName"
></UiInput>
<UiInput
v-if="typeOfCooperation === EJoinUsTypeOfCooperationRadios.COMPANY ||
typeOfCooperation === EJoinUsTypeOfCooperationRadios.PARTNER"
:required="true"
:placeholder="t('applyToJoin.companySite')"
:light="viewPortStore.width <= BREAKPOINTS.PHONE"
:invalid="!!companySiteError"
:errorMessage="companySiteError"
```

```
v-model.trim="companySite"
></UiInput>
<UiInput
v-if="typeOfCooperation === EJoinUsTypeOfCooperationRadios.COMPANY ||
typeOfCooperation === EJoinUsTypeOfCooperationRadios.PARTNER"
:required="true"
:placeholder="t('applyToJoin.jobTitle')"
:light="viewPortStore.width <= BREAKPOINTS.PHONE"
:invalid="!!jobTitleError"
:errorMessage="jobTitleError"
v-model.trim="jobTitle"
></UiInput>
<UiInput
:required="true"
:placeholder="t('shared.email')"
:light="viewPortStore.width <= BREAKPOINTS.PHONE"
:invalid="!!emailError"
:errorMessage="emailError"
v-model.trim="email"
></UiInput>
<UiInput
v-if="typeOfCooperation === EJoinUsTypeOfCooperationRadios.PROFESSIONAL"
:required="true"
:placeholder="t('applyToJoin.yourLinkedInFacebookPortfolio')"
:light="viewPortStore.width <= BREAKPOINTS.PHONE"
:invalid="!!linkOnProfileError"
:errorMessage="linkOnProfileError"
v-model.trim="linkOnProfile"
></UiInput>
</div>
<!-- TODO: Please confirm your consentPlease confirm your consent -->
<UiCheckbox
id="applyToJoinFormAgreementCheckbox"
class="apply-to-join-page__form-checkbox"
:required="true"
```

```
:light="viewPortStore.width <= BREAKPOINTS.PHONE"
:errorMessage="receiveEmailsAgreementError"
v-model="receiveEmailsAgreement"
>
{{ t('shared.iAgreeToReceiveEmailsFromXrUa.part1') }}
<strong>
{{ t('shared.iAgreeToReceiveEmailsFromXrUa.part2') }}
</strong>{{ t('shared.iAgreeToReceiveEmailsFromXrUa.part3') }}
</UiCheckbox>
<div class="apply-to-join-page__form-confirm">
<UiButton
class="apply-to-join-page__form-button"
:design="EButtonDesign.PRIMARY"
:size="EButtonSize.LG"
:light="viewPortStore.width <= BREAKPOINTS.PHONE"
:isLoading="isRequestInProgress"
>
{{ t('applyToJoin.sendRequest') }}
<template #icon>
<IconsArrowRightAnimated />
</template>
</UiButton>
<p class="apply-to-join-page__form-message">
{{ t('shared.byClickingTheSendRequestButton.part1', { buttonName:
t('applyToJoin.sendRequest') }) }}
<NuxtLink :to="{ name: ROUTE_NAMES.PRIVACY_POLICY }" class="apply-to-join-
page__form-message-link" target="_blank">
{{ t('shared.byClickingTheSendRequestButton.part2') }}
</NuxtLink>
</p>
</div>
</form>
</UiModal>
<UiPopup
:isOpen="isSuccessPopupOpen"
```

```
:title="t('shared.ready') + '!'"
:message="t('shared.thanksABunchForReachingOut')"
@onClosePopup="closeSuccessPopup"
/>
</div>
</template>
<script setup lang="ts">
import * as yup from 'yup';
import { EButtonDesign, EButtonSize, EJoinUsTypeOfCooperationRadios, type IJoinUsFormData } from '~/types';
import { useSendpulseJoinUsFormApi } from '~/api/sendpulse/join-us-form.api';
import { useJoinUsFormApi } from '~/api/strapi/join-us-from.api';
import { getClearObject, hasRequiredFields } from '~/helpers';
import { APPLY_TO_JOIN_QUERIES, BREAKPOINTS, ROUTE_NAMES } from '~/constants';
definePageMeta({
layout: 'empty',
});
const { t } = useI18n();
const { safeRouterBack } = useHelpers();
const { sendApplyToJoinMailToUser, sendApplyToJoinMailToAdmin } = useSendpulseJoinUsFormApi();
const route = useRoute();
const router = useRouter();
const notificationsStore = useNotificationsStore();
const viewPortStore = useViewPortStore();
const isSuccessPopupOpen = ref(false);
const isRequestInProgress = ref(false);
const typeOfCooperation = computed<EJoinUsTypeOfCooperationRadios>({
get() {
const typeOfCooperationQuery = route.query[APPLY_TO_JOIN_QUERIES.COOPERATION_TYPE];
const queryValue = typeof typeOfCooperationQuery === 'string' ? typeOfCooperationQuery : undefined;
```

```
return (queryValue && Object.values(EJoinUsTypeOfCooperationRadios).includes(queryValue as EJoinUsTypeOfCooperationRadios))
? (queryValue as EJoinUsTypeOfCooperationRadios)
: EJoinUsTypeOfCooperationRadios.COMPANY;
},
set(value) {
const current = route.query[APPLY_TO_JOIN_QUERIES.COOPERATION_TYPE];
if (current !== value) {
router.replace({
query: { ...route.query, [APPLY_TO_JOIN_QUERIES.COOPERATION_TYPE]: value }
});
}
},
});
const validationSchema = computed(() => {
return yup.object({
fullName: yup
.string()
.required(t('validation.required'))
.noLeadingSpacesOrSymbols(t('validation.cantStartEndWithSpecialCharacters'))
.noTrailingSpacesOrSymbols(t('validation.cantStartEndWithSpecialCharacters'))
.min(2, t('validation.minLength', { min: 2 }))
.max(50, t('validation.maxLength', { max: 50 }))
.isBasicText(t('validation.invalidCharacters')),
email: yup
.string()
.required(t('validation.required'))
.noLeadingSpacesOrSymbols(t('validation.cantStartEndWithSpecialCharacters'))
.noTrailingSpacesOrSymbols(t('validation.cantStartEndWithSpecialCharacters'))
.max(255, t('validation.maxLength', { max: 255 }))
.isValidEmail(t('validation.invalidEmailFormat')),
receiveEmailsAgreement: yup
.boolean()
.oneOf([true], t('validation.pleaseConfirmYourConsent')),
```

```
...((typeOfCooperation.value === EJoinUsTypeOfCooperationRadios.COMPANY ||
typeOfCooperation.value === EJoinUsTypeOfCooperationRadios.PARTNER) ? {
companyName: yup
.string()
.required(t('validation.required'))
.min(2, t('validation.minLength', { min: 2 }))
.max(100, t('validation.maxLength', { max: 100 }))
.isValidCompanyName(t('validation.invalidCharacters')),
companySite: yup
.string()
.required(t('validation.required'))
.max(2000, t('validation.maxLength', { max: 2000 }))
.isValidUrl(t('validation.invalidLinkFormat')),
jobTitle: yup
.string()
.required(t('validation.required'))
.noLeadingSpacesOrSymbols(t('validation.cantStartEndWithSpecialCharacters'))
.noTrailingSpacesOrSymbols(t('validation.cantStartEndWithSpecialCharacters'))
.min(2, t('validation.minLength', { min: 2 }))
.max(100, t('validation.maxLength', { max: 100 }))
.isBasicText(t('validation.invalidCharacters')),
} : {}),
...((typeOfCooperation.value === EJoinUsTypeOfCooperationRadios.PROFESSIONAL)
? {
linkOnProfile: yup
.string()
.required(t('validation.required'))
.max(2000, t('validation.maxLength', { max: 2000 }))
.isValidUrl(t('validation.invalidLinkFormat')),
} : {}),
});
});
const { handleSubmit, resetField, setFieldError, resetForm } = useForm({
validationSchema,
initialValues: {
```

```
fullName: '',
email: '',
companyName: '',
companySite: '',
jobTitle: '',
linkOnProfile: '',
receiveEmailsAgreement: true,
},
});
const { fillJoinUsForm } = useJoinUsFormApi();
const { value: fullName, errorMessage: fullNameError } = useField<string>('fullName');
const { value: email, errorMessage: emailError } = useField<string>('email');
const { value: companyName, errorMessage: companyNameError, handleReset: companyNameReset } = useField<string>('companyName');
const { value: companySite, errorMessage: companySiteError, handleReset: companySiteReset } = useField<string>('companySite');
const { value: jobTitle, errorMessage: jobTitleError, handleReset: jobTitleReset } = useField<string>('jobTitle');
const { value: linkOnProfile, errorMessage: linkOnProfileError, handleReset: linkOnProfileReset } = useField<string>('linkOnProfile');
const { value: receiveEmailsAgreement, errorMessage: receiveEmailsAgreementError } = useField<boolean>('receiveEmailsAgreement');
const onSubmit = handleSubmit(async (values) => {
try {
isRequestInProgress.value = true;
const payload: IJoinUsFormData = {
type: typeOfCooperation.value,
full_name: values.fullName,
email: values.email,
company_name: values.companyName,
company_site: values.companySite,
job_title: values.jobTitle,
profile_link: values.linkOnProfile,
};
const clearPayload = getClearObject(payload);
```

```
if (hasRequiredFields(clearPayload, ['full_name', 'email'])) {
await fillJoinUsForm(clearPayload);
} else {
throw new Error('Illia played too much with types');
}
await sendApplyToJoinMailToUser(
{
type: clearPayload.type,
full_name: clearPayload.full_name,
email: clearPayload.email,
},
);
await sendApplyToJoinMailToAdmin(clearPayload);
resetForm();
isSuccessPopupOpen.value = true;
} catch (error) {
notificationsStore.add({
type: 'error',
title: t('notifications.sorry'),
message: t('notifications.weCouldNotSendYourData'),
});
} finally {
isRequestInProgress.value = false;
}
});
function closeModal() {
safeRouterBack();
}
async function closeSuccessPopup() {
isSuccessPopupOpen.value = false;
router.push({ name: ROUTE_NAMES.INDEX });
}
watch(typeOfCooperation, (newValue) => {
if (newValue === EJoinUsTypeOfCooperationRadios.PROFESSIONAL) {
companyNameReset();
```

```
companySiteReset();
jobTitleReset();
} else if (newValue === EJoinUsTypeOfCooperationRadios.COMPANY || newValue ===
EJoinUsTypeOfCooperationRadios.PARTNER) {
linkOnProfileReset();
}
});
</script>
<style lang="scss">
.apply-to-join-page {
&__form {
display: flex;
flex-direction: column;
gap: 28px;
margin-block-end: 16px;
}
&__radios {
display: flex;
flex-direction: column;
gap: 16px;
}
&__inputs {
display: flex;
flex-direction: column;
gap: 28px;
margin-block-end: 20px;
}
&__form-checkbox {
margin-block-end: 20px;
}
&__form-button {
width: 100%;
margin-block-end: 16px;
}
&__form-message {
```

```
font-size: 12px;
line-height: 1.4;
@include phone {
color: $gray-50;
}
}
&__form-message-link {
text-decoration: underline;
color: $blue;
@include phone {
color: $aqua;
}
}
}
</style>
```

Додаток Є

```vue
<template>
  <div
    class="ui-tag"
    :class="{
      'ui-tag--secondary': design === 'secondary',
      'ui-tag--outlined': outlined,
    }"
  >
    <span class="ui-tag__label">
      {{ label }}
    </span>
  </div>
</template>

<script setup lang="ts">
interface Props {
  label: string;
  design?: 'primary' | 'secondary';
  closable?: boolean;
  outlined?: boolean;
}

withDefaults(defineProps<Props>(), {
  design: 'primary',
  closable: false,
  outlined: false,
});
</script>

<style lang="scss">
.ui-tag {…
}
</style>
```

Рисунок 1 – Фрагмент коду простого Vue-компонента, створеного для багаторазового використання у клієнтській частині веб-застосунку

```ts
useHead({
  title: 'XR Ukraine Association — Unites Innovators in Immersive Tech',
  meta: [
    {
      name: 'description',
      content: 'XR Ukraine Association brings together AR/VR/MR companies and experts with Ukrainian roots, fostering innovation and connecting to the global immersive tech ecosystem.',
    }
  ],
  style: […
  ],
});
```

Рисунок 2 – Фрагмент коду для налаштування заголовку та метаданих сторінки у Nuxt 3 за допомогою функції useHead() з метою покращення SEO-показників веб-застосунку